ELSEVIER
The Journal of Logic Programming 41 (1999) 67–102
THE JOURNAL OF LOGIC PROGRAMMING
www.elsevier.com/locate/jlpr

A pragmatic reconstruction of λProlog

Catherine Belleannée [a], Pascal Brisset [b,1], Olivier Ridoux [a,2,*]

[a] *IRISA, Campus universitaire de Beaulieu, F-35042 Rennes Cedex, France*
[b] *ENAC, 7 av. Édouard Belin, BP 4005, F-31055 Toulouse Cedex 4, France*

Received 19 October 1995; received in revised form 25 February 1998; accepted 30 September 1998

Abstract

λProlog is a logic programming language in which hereditary Harrop formulas generalize Horn formulas, and simply typed λ-terms generalize Prolog terms. One may wonder if these extensions are simultaneously required, and if useful subsets of λProlog can be defined, at least for pedagogical purposes. We answer this question by exhibiting a network of necessity links between the new features of λProlog. The starting point of the network is the demand for programming by structural induction on λ-terms, and the necessity links give a rationale for such a programming style.

Keywords: λProlog; Reconstruction; Structural induction

1. Introduction

Logic programming is a programming paradigm in which programs are logical formulas, and executing them amounts to searching for a proof. A logic programming language is usually a syntactical fragment of some logic that has been chosen for its appeal to programming intuition. The most famous practical logic programming language is Prolog, which is based on Horn formulas [30].

The formalism of Horn programs is computationally complete [1,59], but it has often been tried to augment it to gain more flexibility and expressiveness. One of these attempts is λProlog [40,42]. It preserves a simple formal connection to logic, which is rare among extensions to Horn formulas. Indeed, a kind of goal directed proofs (proofs that can be used as an operational semantics) is complete for the formulas of λProlog.

The following equation sketches the definition of λProlog:

$$\lambda Prolog \stackrel{\text{def}}{=} Prolog + \ \lambda\text{-}terms + \ simple\ types + \ =_{\alpha\beta\eta} + \forall_{\mathcal{G}} + \Rightarrow_{\mathcal{G}}$$

* Corresponding author. Tel.: +33 2 99 847330; fax: +33 2 99 847171; e-mail: ridoux@irisa.fr.
[1] E-mail: ridoux@irisa.fr.
[2] E-mail: brisset@recherche.enac.fr.

0743-1066/99/$ – see front matter
PII: S0743-1066(98)10038-9

The components of this formula will be completely defined in the sequel. For now, it is enough to know that λ-*terms*, *simple types*, and $=_{\alpha\beta\eta}$ deal with the computing domain, while $\forall_{\mathscr{G}}$, and $\Rightarrow_{\mathscr{G}}$ deal with the structure of programs. The subscripted $\mathscr{G}$ means that $\forall$ and $\Rightarrow$ can be used as constructors of goals (clause bodies). Similarly, a subscripted $\mathscr{D}$ means that the connective is used as a constructor for definite clauses. For instance, Horn clauses admit conjunctions in their bodies (i.e., $\wedge_{\mathscr{G}}$), and Horn clauses are constructed with the implication connective (i.e., $\Rightarrow_{\mathscr{D}}$, see more details on this in Section 2.1.4).

Possible applications of λProlog are chiefly the applications that motivated the introduction of λ-terms [46,40]: manipulation of formulas, computation of denotations, etc. (see Section 3.1). Notice also that the structure of λProlog encompasses such constructions as modules [35] and abstract data-types [33] without any extralogical additions. There have been actual applications of λProlog for automatic theorem proving [3,15], analysis of natural and formal languages [52,11,28,56], and the manipulation of functional programs [21]. Note also applications which mix several technologies like the recognition of musical scores (pixel-level and symbolic-level analyses) [10].

Beside the initial demonstration implementation of λProlog, three implementations of λProlog can be used: in the chronological order, eLP is an interpreter written in Lisp [14], Prolog/Mali is a compiler written in λProlog which generates C programs [5], and Terzo is an interpreter written in ML. A resource page, *http://www.cse.psu.edu/~dale/lProlog/*, gives access to these implementations and to the literature.

The first steps of a newcomer to λProlog are difficult because the relationships between all the components of λProlog are complex, and it is not clear how to use them. We aim at giving an explanation of the relationships via a pragmatic reconstruction of λProlog. An expected by-product is a rationale for a class of λProlog programs.

The proposed construction is not the only possible for λProlog (see for instance the texts of its designers), but it is one that spans all the features of λProlog from one point of view, and that supports a useful class of λProlog programs. The texts by the designers and first users of λProlog often propose different points of view (e.g., higher-order programming, modularity, theorem proving, data abstraction) for describing the different features of λProlog. They often show how λProlog satisfies several requirements, but not a single and general requirement that justifies the whole of λProlog. In the beginning, Miller and Nadathur present a logic programming language called λProlog, which features higher-order Horn clauses, λ-terms, and λ-equivalence [40,46]. Then, Miller formalizes module importation as logical

Table 1
A bibliography map

	1986	1987	1988	1989	1990	1991	1992	1993
Higher-order LP	[40]	[41,46]			[47]			
Modules	[37]			[35]				[36]
$\Rightarrow_{\mathscr{G}}$	[37]	[41,43]				[42]		
$\forall_{\mathscr{G}}$		[43]		[33]		[42]		
Decidable higher-order unification				[34]		[38]		
Abstract syntax						[32]		
Unification and quantification							[39]	

implication in goals, $\Rightarrow_{\mathcal{G}}$ [35], and module abstraction as universal quantification in goals, $\forall_{\mathcal{G}}$ [33]. Miller et al. observe that these extensions form a well-behaved fragment of intuitionistic logic [43,42]. This fragment is called *hereditary Harrop formulas*, and the extended language is still called λProlog. Table 1 sums-up the introduction of the basic concepts of λProlog in its creators' writings.

To ease the way of a beginner, it is tempting to define fragments of λProlog by merely dropping some of its components. For instance,

$$\textit{Typed Prolog} \stackrel{\text{def}}{=} \textit{Prolog} + \textit{simple types}$$

defines a strongly typed variant of Prolog as proposed by Lakshman and Reddy [27].

$$\textit{CLP}(\lambda_{\rightarrow}) \stackrel{\text{def}}{=} \textit{Prolog} + \lambda\textit{-terms} + \textit{simple types} + =_{\alpha\beta}$$

defines an instance of the scheme CLP [9] for the domain of the simply typed λ-terms endowed with the equivalence relation $=_{\alpha\beta}$ [57].

$$\textit{Harrop Prolog} \stackrel{\text{def}}{=} \textit{Prolog} + \forall_{\mathcal{G}} + \Rightarrow_{\mathcal{G}}$$

defines an extension of Prolog in which connectives $\forall_{\mathcal{G}}$ and $\Rightarrow_{\mathcal{G}}$ are allowed.

$$\alpha\beta\textit{Prolog} \stackrel{\text{def}}{=} \textit{Prolog} + \lambda\textit{-terms} + \textit{simple types} + =_{\alpha\beta} + \forall_{\mathcal{G}} + \Rightarrow_{\mathcal{G}}$$

defines the extension to Prolog that differs from λProlog only in that η-equivalence is not considered. This possibility is left open in an early article on λProlog [40].

Even if interesting programs can be written in all these fragments, they are not of equal value. In this work we consider as the quality criterion the ability of programming by induction on the structure of λ-terms. This allows us to deem each fragment useful, or not, for this purpose. *Structural induction* informally describes the programming tactic of using the structure of the argument of a computation to decide how to decompose it, what treatment to apply to its components, and how to compose the results thus obtained. This is related to formal *induction* by a property of the type of the argument that is called *inductiveness*, but we will show that λProlog makes it possible to program by structural induction on data-structures that are not formally inductive. When comparing fragments we always consider pure ones and we ignore built-in predicates. What is at stake here is the possibility to express structural induction on λ-terms in a logical way.

Our reconstruction assumes that the first step is to introduce λ-terms and $\alpha\beta$-equivalence in Prolog. Then, a more detailed analysis shows that more capabilities are needed. Fig. 1 illustrates the relations between the features of λProlog. An arrow from A to B reads "A makes B possible". The nature of these arrows is the subject of

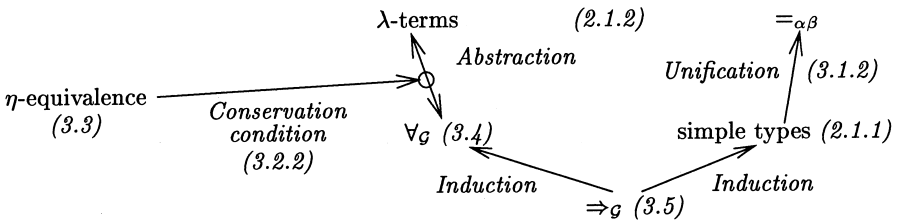


Fig. 1. A structure for λProlog.

this work and will be exposed at length in the sequel (see section numbers in the figure). Yet, we introduce them now in a few words.

Executing logic programs (e.g., in the Prolog sense) with λ-terms and $\alpha\beta$-equivalence requires to solve unification problems modulo $\alpha\beta$-equivalence. Since the unification of λ-terms modulo $\alpha\beta$-equivalence is overly difficult, one must restrict the problem; typing terms makes the problem easier, hence the arrow *Unification*. Programming by structural induction on λ-abstractions requires to be able to relate a λ-abstraction and its body. Since pure logic programming is non-directional, this relation must be symmetric: the same expression can be used to get the body of an abstraction, and to abstract a λ-variable in a term. This can been done by applying λ-abstractions to universally quantified variables. This relation is symmetric only if a conservation condition is met. The η-equivalence axiom entails this condition. Finally, though structural induction implies a closed view of data-structures because all cases must be determined at programming time, and λProlog's universal quantification implies openness because it is interpreted as introducing new constants, both can be reconciled by using implications in goals; hence the two *Induction* arrows.

Note that a capability may have been introduced in λProlog for other reasons that we do not mention here. This results from our bias towards programming by structural induction. Conversely, we do not mention other capabilities that could have been introduced for the same purpose. This is because we describe λProlog and nothing else.

A way to help λProlog beginners is to show them heuristics and idioms for building λProlog programs. Our pragmatic reconstruction produces a heuristic that is based on the consultation of the types of the data-structures that a programmer wants to manipulate. It indicates when to use universal quantification and implication in goals, and it can be formalized in a general structural induction scheme.

We briefly present in Section 2 the syntax and semantics of λProlog (we assume a basic knowledge of Prolog, but no knowledge of λProlog or the λ-calculus). Section 3 contains a necessity-driven reconstruction of λProlog. We then propose a general scheme for programming by structural induction on λ-terms (Section 4). Finally, we show how the reconstruction tallies with an important fragment of λProlog, L_λ, and we briefly present other useful programming idioms (Section 5).

Every program sample will use the concrete syntax of λProlog (in its Prolog/Mali implementation), or of Standard Prolog in the rare cases of Prolog samples.

2. λProlog

Before entering into the details of λProlog let us have a glimpse at what it looks like. λProlog allows us to write programs similar to Typed Prolog programs [27]:

```
(1) kind list type -> type.
(2) type [ ] (list A).
(3) type '.' A -> (list A) -> (list A).
(4) type append (list T) -> (list T) -> (list T) -> o.

(5) append [ ] L L.
(6) append [A / L1] L2 [A / L3]:- append L1 L2 L3.
```

where line (1) introduces a type constructor *list*, lines (2) and (3) introduce constructors *nil* and *cons* (written [ ] and '.' or [/] as in Prolog [1]). Line (4) introduces a constant *append* that forms a proposition when given three suitably typed arguments. Lines (5) and (6) are similar to the Prolog version of *append* except for the omission of parentheses and commas. Otherwise, the same lexical conventions hold in general, and the same logical and operational semantics hold in this example.

λProlog also makes it possible to write programs with a really new structure:

```
typing (abs E) (arrow A B):-pi x \ (typing x A => typing (E x) B).
% typing (abs E) (A → B) ⇐ ∀x[typing x A ⇒ typing (E x) B]
```

in which E is a function variable, *pi* is a universal quantifier, => is an implication connective, and x is universally quantified in the clause body (the text starting with a % is a comment). The reading of this clause is that if for all x of type A the application of E to x has type B, then the λ-abstraction E has type $A \to B$. Its development is presented in Section 3.5.2.

2.1. The principles of λProlog

2.1.1. The types

The new term language is the language of the simply typed [2] λ-terms [8] augmented with variables in types. Simple types are generated by the following grammar. Note that T^i reads "T repeated i times".

$$T ::= \mathscr{U} \mid (\mathscr{K}_i T^i) \mid (T \to T),$$

where the $\mathscr{U}$'s and $\mathscr{K}_i$'s are respectively type variables and type constructors with arity i. We assume that $\mathscr{K}_0$ contains at least the constant 'o' for propositions. Types like $(\mathscr{K}_i\ T^i)$ are called *base types*. The third alternative generates *function types* (also called *arrow types*). A type $(A \to B)$ can be interpreted as the type of functions whose domain is A and codomain is B. We also assume the arrow associates to the right. This makes some brackets useless. For instance, $o \to o \to o$ denotes the same type as $(o \to (o \to o))$ does. The concrete syntax for '→' is '– >'.

In λProlog, type constructors are declared using directive *kind*: for instance,

```
kind o type.          % o ∈ K0
kind int type.        % int ∈ K0
kind list type -> type.   % list ∈ K1
```

The declaration of *list* shows it is a type constructor that must be applied to one type to actually produce a type. So, (list int) is a valid type, but *list* alone is not. Expression $(list\ A) \to (list\ A)$, where A is a variable, is also a valid type. It is the type of functions whose domain and codomain are (*list t*) *for any type t*. These types are very close to ML types [44]. Every variable in a type must be considered as versally quan-

[1] Some implementations of λProlog use *::* as in ML.

[2] The phrase *simple type* should be considered as a proper name. No formal notion of simpleness is involved.

tified at the beginning of the type: e.g., $\forall A[(list\ A) \rightarrow (list\ A)]$. The way types are associated to terms is introduced in the next section.

2.1.2. The terms

Simply typed λ-terms are generated by the following grammar:

$$\begin{array}{lll} \Lambda_{t'\rightarrow t} ::= \lambda \mathscr{V}_{t'} \Lambda_t & & \text{if } t,\ t' \in \mathrm{T} \\ \Lambda_t \quad ::= \mathscr{C}_t \mid \mathscr{V}_t \mid (\Lambda_{t'\rightarrow t} \Lambda_{t'}) & & \text{if } t,\ t' \in \mathrm{T} \end{array}$$

where the $\mathscr{C}_t$'s and $\mathscr{V}_t$'s are respectively constants and λ-variables whose type is t. A λ-calculus is called *pure* if $\mathscr{C}_t$ is empty for every t. Attributes in terminal and non-terminal symbols are used to constrain types and to ensure the well-typing of generated terms. The first rule generates *λ-abstractions*, and the third alternative of the other rule generates *applications*. A λ-abstraction can be interpreted as a function, and an application can be interpreted as the result of applying a function to some actual parameter. We assume application associates to the left. For instance, (*append A B C*) denotes the same term as (((*append A*) *B*) *C*) does. In the concrete syntax, λ-abstraction is written with an infix '\' instead of the classical prefix 'λ'. For instance, the identity function is written $x \setminus x$ instead of $\lambda x(x)$.

A consequence of typing is to prevent the application of λ-variables to themselves (self-application). For instance, $\lambda x(x\ x)$ is not well-typed because of self-application $(x\ x)$. Note that self-application of non-variable terms is allowed as in $(\lambda x(x)\ \lambda x(x))$, but the x in the first $\lambda\ x(x)$ and the x in the second one will have different types.

λ-Abstraction leads to the notions of *heading*, *head*, and *body*. In the term $\lambda a \lambda b \lambda c(b\ a\ c)$, the heading is $\lambda a \lambda b \lambda c$, the head is b, and the body is $(b\ a\ c)$. *One says a term binds* the variables of its heading. One also distinguishes between *free* and *bound* occurrences of λ-variables. For instance, in the term $(x\ y\ \lambda z(\underline{\lambda x(x\ y\ z)}))$, y has only free occurrences, the only occurrence of z is bound, and x has both a free occurrence and a bound one (the first and the second, respectively). In the underlined subterm, z and y have only free occurrences, and the only occurrence of x is bound. More generally, an occurrence of some λ-variable is bound in a term if it is a subterm of a λ-abstraction that binds the λ-variable and is a subterm of the term. An occurrence of some λ-variable is free if it is not bound. One calls *free variables* of a term t, $\mathscr{FV}(t)$, the variables that have a free occurrence in it, *bound variables*, $\mathscr{BV}(t)$, those that have a bound occurrence in the term. A term without any free occurrences of a λ-variable is called a *closed* term or a *combinator*. One writes $[x \leftarrow y]$ for the operation of replacing all free occurrences of x by y, and $E[x \leftarrow y]$ for its application to E.

In λProlog, constants and their types are declared using directive *type*:

$$\begin{array}{l} type\ [\ ](list\ T). \qquad \%\ \text{nil:}\forall \mathrm{T}[[\ ] \in \mathscr{C}_{(\text{list T})}] \\ type\ '.'\ T->(list\ T)->(list\ T). \\ \qquad\qquad \%\,\text{cons:}\forall \mathrm{T}['.' \in \mathscr{C}_{T\rightarrow(listT)\rightarrow(listT)}] \\ type\ append\ (list\ T)->(list\ T)->(list\ T)->o. \\ \qquad\qquad \%\forall \mathrm{T}[\text{append} \in \mathscr{C}_{(\text{list T})\rightarrow(\text{list T})\rightarrow(\text{list T})\rightarrow \text{o}}] \end{array}$$

Given a program, the collection of all pairs $\langle c, \tau \rangle$ such that the program contains a declaration *type* $c\ \tau$ forms the *signature* of the program. The type of [ ] shows it is a

non-functional constant. The type of '.' shows it is a functional constant that takes two arguments. These two constants allow us to build polymorphic lists, but the declared types force all the elements of a given list to have the same type. The lists that have this type are called *homogeneous*. Finally, the result type of *append, 'o'*, shows it is a propositional function.

Note that the notion of arity is rendered by the number of arrows in the type, but since λProlog is a higher-order programming language nothing forces to apply a constant to as many arguments as its arity. [3]

2.1.3. The equality theory

The domain of simply typed λ-terms is endowed with an equivalence relation which is defined as the smallest congruence based on the following axioms [2].

Axiom α: $\lambda x(E) =_\alpha \lambda y(E[x \leftarrow y])$, if $y \in \mathscr{V}$ and $y \notin \mathscr{FV}(\lambda x(E)) \cup \mathscr{BV}(E)$.

This axiom formalizes the renaming of bound λ-variables. The side condition is for preventing capture of variables. For instance, $\lambda x(f\ x) =_\alpha \lambda y(f\ y)$ but $\lambda x(g\ x\ y) \neq_\alpha \lambda y(g\ y\ y)$ because $y \in \mathscr{FV}(\lambda x(g\ x\ y))$. In the second example, y would be captured by the renaming of x.

Axiom β: $(\lambda x(E)F) =_\beta E[x \leftarrow F]$, if $\mathscr{FV}(F) \cap \mathscr{BV}(E) = \emptyset$.

This axiom formalizes the evaluation of an application by substituting an actual parameter, F, for a formal parameter, x. Again, the side condition is for preventing capture of variables. For instance, $(\lambda x(f\ x)12) =_\beta (f\ 12)$ but $(\lambda x \lambda y(x)\ y) \neq_\beta \lambda y(y)$ because $y \in \mathscr{FV}(y)$ and $y \in \mathscr{BV}(\lambda x \lambda y(x))$.

Axiom η: $\lambda x(E\ x) =_\eta E$, if $x \notin \mathscr{FV}(E)$.

This axiom entails *functional extensionality* in the pure λ-calculus: "Functions with undistinguishable results are equal." According to this principle, we have $\forall x[(E\ x) = (F\ x)] \Rightarrow E = F$, which is not provable using axioms α and β alone.

For instance, $\lambda x(f\ x) =_\eta f$, but $\lambda x((g\ x)x) \neq_\eta (g\ x)$ because $x \in \mathscr{FV}((g\ x))$.

We will write $=_x$ any such equivalence relation based on a subset x of axioms α, β, and η. Note that $(\lambda x \lambda y(x)y) =_\alpha (\lambda x \lambda w(x)y) =_\beta \lambda w(y)$. In fact, it is always possible to apply axiom β to a term like $(\lambda x(E)F)$ if one applies axiom α for renaming the bound variables of E. As a result, axiom α is often applied silently with axiom β. On the other hand, axiom α can never help applying axiom η.

The pattern $(\lambda x(E)F)$ is called a *β-redex*. A λ-term with no β-redex is said to be in *normal form*. A λ-term whose head is a variable or a constant is said to be in *head-normal form*, even if it contains β-redexes. Converting λ-terms into λ-equivalent

[3] This way of representing n-ary functions by cascading unary functions is called *currying* after H.B. Curry, although it can be traced back to J. Schönfinkel and G. Frege.

(head-) normal forms is important for comparing them. However, in the pure λ-calculus, a λ-term may not be λ-equivalent to any normal form, whereas in the simply typed λ-calculus, a λ-term is always λ-equivalent to some normal form. For instance, $(\lambda x(x\ x)\lambda x(x\ x))$ is not normal because it is a β-redex, but it is only β-equivalent to itself, so it has no normal form. However, this term cannot be simply typed because of the self-application $(x\ x)$.

Axioms β and η can be oriented to form rewrite rules. They are called *reduction* rules when oriented from left to right, and *expansion* rules otherwise. One forms this way β-reduction, η-reduction, and η-expansion. β-Expansion is not determined enough to be considered as a rewrite rule, but it can be related to unification. A term that contains several β-redexes can be the starting point of many reduction sequences. However, the λ-calculus has the Church–Rosser property: all these sequences can be prolonged to converge to a common term. Moreover, the simply typed λ-calculus has the *strong normalization property*, which says that there is no infinite sequence of β-reductions. As a consequence, all selection strategies for β-redexes are terminating.

The pure untyped λ-calculus is a computability model. All natural numbers can be encoded as pure λ-terms: for instance, (we note $\ulcorner i \urcorner$ the code of i) the Church encoding of 0 is $\lambda s\lambda z(z)$, $\ulcorner 1 \urcorner$ is $\lambda s\lambda z(s\ z)$, $\ulcorner 2 \urcorner$ is $\lambda s\lambda z(s(s\ z))$, and more generally, $\ulcorner i \urcorner$ is $\lambda s\lambda z(s^i\ z)$. Many data-structures (e.g., pairs, lists, trees) can also be encoded as pure λ-terms. Every computable function f on integers can be encoded as a pure λ-term $\ulcorner f \urcorner$ such that $(\ulcorner f \urcorner\ \ulcorner i \urcorner)$ λ-reduces to $\ulcorner f(i) \urcorner$. This is no longer true for simply typed terms. The lack of self-application of variables forbids to build fix-point operators, which are essential to model general recursion and to have a complete computation model. So, λProlog's terms do not form a computationally complete functional language.

Note also that axiom η entails functional extensionality only in the pure λ-calculus. In the simply typed λ-calculus, two functions might be undistinguishable for the applications that are well-typed, though they are not η-equivalent. For instance, $\lambda n\lambda s\lambda z(s(n\ s\ z))$ and $\lambda n\lambda s\lambda z(n\ s(s\ z))$ are two possible encodings of the successor function for natural numbers in their Church encoding. The first one adds a leading s to an n, and the second one substitutes $(s\ z)$ to the trailing z of n. These two λ-terms are not η-equivalent, but they cannot be distinguished when they are applied to pure terms of type $(\alpha \to \alpha) \to \alpha \to \alpha$.

Axioms α and β are always present in the λ-calculus, but axiom η is optional: it does not affect the computational completeness of the λ-calculus. However, we will see that axiom η is crucial for the equality theory.

2.1.4. The formulas

Programs, clauses and goals are generated by the following grammar:

$$
\begin{aligned}
\mathscr{P} &::= \mathscr{D} \mid \mathscr{D} \wedge_{\mathscr{D}} \mathscr{P} \qquad \text{if } \mathscr{FV}(\mathscr{D}) = \emptyset \\
\mathscr{D} &::= \mathscr{A} \mid \mathscr{A} \Leftarrow_{\mathscr{D}} \mathscr{G} \mid \forall_{\mathscr{D}} \mathscr{V}(\mathscr{D}) \\
\mathscr{G} &::= \mathscr{A} \mid \mathscr{G} \wedge_{\mathscr{G}} \mathscr{G} \mid \mathscr{G} \vee_{\mathscr{G}} \mathscr{G} \mid \mathscr{D} \Rightarrow_{\mathscr{G}} \mathscr{G} \mid \forall_{\mathscr{G}} \mathscr{V}(\mathscr{G}) \mid \exists_{\mathscr{G}} \mathscr{V}(\mathscr{G}) \\
\mathscr{A} &::= \Lambda_{\mathrm{o}}
\end{aligned}
$$

Rules $\mathscr{P}$, $\mathscr{D}$, $\mathscr{G}$and $\mathscr{A}$ generate respectively *programs*, *clauses*, *goals* and *atomic formulas* (also called *atoms*). The novelty of λProlog is in the goal language: explicit

quantifications (universal and existential) and implications may occur in goals. Formulas generated this way are called *hereditary Harrop formulas*. [4] The connectives $\vee_{\mathscr{G}}$ and $\exists_{\mathscr{G}}$ can be defined using the other connectives in higher-order Horn clauses. However, we mention them because they ease the writing of programs.

In the concrete syntax, connective $\wedge_{\mathscr{D}}$ is implicit, and $\Leftarrow_{\mathscr{D}}$, $\wedge_{\mathscr{G}}$, $\vee_{\mathscr{G}}$ are written ':–', ',' and ';' like in Prolog. Since program clauses contain no free variable (see Rule $\mathscr{P}$), every variable left free in the concrete syntax of a program clause is considered bound by an implicit $\forall_{\mathscr{D}}$. Variables bound by implicit $\forall_{\mathscr{D}}$'s are distinguished from constants in $\mathscr{C}_t$ by the usual Prolog convention: names of variables start with a capital letter or an underscore ('_'). Connectives $\Rightarrow_{\mathscr{G}}$, $\forall_{\mathscr{G}}$ (and $\forall_{\mathscr{D}}$, when it is explicit), $\exists_{\mathscr{G}}$ are written '=>', *pi*, and *sigma*. [5] Variables bound by $\forall_{\mathscr{D}}$ or $\exists_{\mathscr{G}}$ are called *existential*, and those bound by $\forall_{\mathscr{G}}$ are called *universal*.

2.1.5. The semantics

The semantics of λProlog is usually given in proof-theoretic terms [42], as opposed to the model-theoretic semantics used for Prolog [30]. There is a model theory for λProlog [42], but it does not designate a preferred model which is as intuitive as the least Herbrand model for Prolog. The main result of the proof-theoretic semantics of λProlog is that a class of goal-directed proofs, called *uniform proofs*, is complete with respect to intuitionistic provability for hereditary Harrop formulas. In other words, every hereditary Harrop formula that is a theorem in intuitionistic logic has a uniform proof.

The deduction rules of the intuitionistic sequent calculus that correspond to connectives $\exists_{\mathscr{G}}$, $\forall_{\mathscr{G}}$ and $\Rightarrow_{\mathscr{G}}$ are as follows [17]:

$$\frac{P \vdash G[x \leftarrow t]}{P \vdash \exists x(G)} \exists_{\mathscr{G}} \text{ (i.e., } \textit{sigma}) \quad t \text{ is an arbitrary term.}$$

$$\frac{P \vdash G[x \leftarrow c]}{P \vdash \forall x(G)} \forall_{\mathscr{G}} \text{ (i.e., } \textit{pi}) \quad c \text{ does not occur free in } P \text{ and G.}$$

$$\frac{P, D \vdash G}{P \vdash D \Rightarrow G} \Rightarrow_{\mathscr{G}} \ (i.e., =>)$$

A sequent $P \vdash G$ reads "goal G is a consequence of program P". A rule

$$\frac{Sequent^*}{Sequent}$$

reads "conclusion *Sequent* is true if all premises *Sequent** are true". All these rules are right introduction rules; their connective of interest is in the right part of the conclusion sequent (i.e., in the goal). These rules, and others that belong to the Horn clause fragment of λProlog, can be used to form proof trees.

---

[4] The name *Harrop formula* refers to R. Harrop's works on conditions such that a formula $A \Rightarrow B \vee C$ (resp. $A \Rightarrow \exists x B(x)$) is provable if and only if either $A \Rightarrow B$ or $A \Rightarrow C$ is provable (resp. $A \Rightarrow B(t)$ is provable for some t) [22]. Harrop formulas have this property at the top-level, and hereditary Harrop formulas have it at any level. This property is important to give a computational meaning to logical formulas. This work had been developed before logic programming as was Horn's work [24] founding the notion of Horn clauses.

[5] The names *sigma* and *pi* date from Peirce's introduction of existential and universal quantifications as generalized sums and products, Σ and Π [53]. This notation has been in use since that time in several works of interest for λProlog and logic programming [8,24].

The operational semantics of the new connectives is as follows:

Connective $\exists_{\mathscr{G}}$: To prove a goal $\exists x(G)$, prove goal $G[x \leftarrow V]$, where V is a new free existential variable which has the type of x.

Connective $\forall_{\mathscr{G}}$: To prove a goal $\forall x(G)$, prove goal $G[x \leftarrow c]$, where c is a new constant which has the type of x, taking care that c does not occur in the binding values of older free existential variables.

Connective$\Rightarrow_{\mathscr{G}}$: To prove a goal $D \Rightarrow G$, add clause D to the program and prove goal G. Clause D is kept in the program during the proof of G. It is suppressed from it as soon as the proof of G is over.

The introduction of free existential variables for handling connective $\exists_{\mathscr{G}}$ leads to the following notions. A term without any free occurrences of existential variables is called *ground*. A head-normal form whose head is either a constant, a λ-variable, or a universal variable is called *rigid*. Other head-normal forms (i.e., whose head is an existential variable) are called *flexible*. The names rigid and flexible come from the fact that a flexible term can take almost any shape via substitution, though a rigid term cannot. For instance, assuming that x, y and A have types $int \rightarrow int$, int and $(int \rightarrow int) \rightarrow int \rightarrow int$, the flexible term $\lambda x \lambda y(A\ x\ y)$ can become $\lambda x \lambda y(x\ y)$ *via* substitution $[A \leftarrow \lambda x(x)]$, $\lambda x \lambda y(33)$ *via* substitution $[A \leftarrow \lambda x \lambda y(33)]$, and $\lambda x \lambda y(+(x\ 33)y)$ *via* substitution $[A \leftarrow \lambda x(+(x\ 33))]$.

Given a node of a proof tree, we call *current signature* the signature of the program (see Section 2.1.2) plus all pairs $\langle c, typeof(c)\rangle$ such that c is introduced between the root of the tree and the given node for interpreting a $\forall_{\mathscr{G}}$. Each c is called a *universal constant*.

The deduction rules and the operational semantics are related as follows. Rule $\exists_{\mathscr{G}}$ is implemented by leaving free a new existential variable that will be bound lazily by unification, instead of guessing a t. The side condition of rule $\forall_{\mathscr{G}}$ is implemented safely by always choosing a new constant. The lazy choice of t in the operational counterpart of rule $\exists_{\mathscr{G}}$ makes the implementation of $\exists_{\mathscr{G}}$ non-atomic, and forces the implementation of the other rules to be cautious about it: hence the constraint on binding values.

An example shows how these operational rules co-operate. Consider a goal $\forall x \exists y[p\ y \Rightarrow p\ x]$. Rule $\forall_{\mathscr{G}}$ applies first; it produces a goal $\exists y[p\ y \Rightarrow p\ c]$ where c must not occur in the bindings of older free existential variables. Since there is no older free existential variables this is a vacuous constraint. Then, rule $\exists_{\mathscr{G}}$ produces a goal $(p\ V \Rightarrow p\ c)$ where V is a new free existential variable. Then, rule $\Rightarrow_{\mathscr{G}}$ is applied and produces a goal $(p\ c)$ while adding clause $(p\ V)$ to the program. Finally, the goal $(p\ c)$ is successfully unified with the head of the added clause; a substitution $[V \leftarrow c]$ is produced. The initial goal is thus proved.

Consider now the same goal with permuted quantifications: $\exists y \forall x[p\ y \Rightarrow p\ x]$. The rule for $\exists_{\mathscr{G}}$ applies first; it produces a goal $\forall x(p\ V \Rightarrow p\ x)$ where V is a new free existential variable. Then, the rule for $\forall_{\mathscr{G}}$ applies and produces a goal $(p\ V \Rightarrow p\ c)$ where c must not occur in the bindings of older free existential variables (i.e., V). The rule for $\Rightarrow_{\mathscr{G}}$ is applied and produces a goal $(p\ c)$ while adding clause $(p\ V)$ to the program. Finally, the unification of $(p\ c)$ and $(p\ V)$ is attempted, but it fails because c must not occur in V.

Despite the apparent shift from classical logic to intuitionism, the logic of λProlog is a conservative extension of that of Prolog. This is because classical logic and intuitionistic logic coincide for Horn theories.

There are two senses in which λProlog is a higher-order language. First, the computation domain is made of λ-terms. This introduces the power of manipulating scoped structures, but it does not make λProlog higher-order in a logical sense. Most of our examples are of this kind. Second, λ-terms may have types that contain type *o*. This introduces the capability of quantifying over propositions or predicates, which makes the logic of λProlog logically higher-order. However, this higher-orderness must be used cautiously, and its application is restricted to formulas built with connectives ∨, ∧, and ∃ [40]. Furthermore, what seems higher-order at first sight, e.g., formulas manipulating formulas, is often meta-programming, i.e., meta-level formulas manipulating object-level formulas. In this latter case, the types of the meta-level formulas and of the object-level formulas can often be kept separate. Most of our examples insist on this point (see types *formula* in Section 3.4, *l_term* in Section 3.2.1).

2.1.6. Three built-in notions of scope

The word "scope" sums-up the new constructs of λProlog:

1. λ-abstractions limit the scope of variables in terms.
2. Quantifications limit the scope of variables in formulas.
3. The deduction rules for universal quantification and implication limit the scope of constants and clauses, respectively, in the proof process.

Point 1 has to do with programming techniques like the manipulation of scoped data-structures and abstract syntax [41,32]. Point 2 offers a refinement over the clause level scoping of Prolog. Point 3 has to do with modularity and information hiding [37,33], but also with the manipulation of scoped data-structures introduced by point 1. We will show in Sections 3 and 4 that programming in λProlog often amounts to making the three scoping levels interact.

2.2. An example

We illustrate the novelties of λProlog by elaborating on the *grandfather* program.

2.2.1. A genealogical data-base in Prolog

Let the predicates *father* and *mother* represent the father–child and mother–child relations in a Prolog data-base.

father(*adam*, *cain*). *father*(*adam*, *abel*). ...
mother(*eve*, *cain*). *mother*(*eve*, *abel*). ...

2.2.2. Introduction of λProlog syntax

To transform the above program into a λProlog program, one must declare the type of every constant, and adopt λProlog syntax (see Section 2.1).

kind person type.
type (*adam, eve, cain, abel, ...*) *person.*
type (*mother, father, ...*) *person* − > *person* − > *o.*

father adam cain. *father adam abel.* ...
mother eve cain. *mother eve abel.* ...

Relations *parent* and *grandfather* can be defined as follows:

```
type (parent, grandfather) person -> person -> o.

parent P C :- father P C.
parent P C :- mother P C.

grandfather GF GC :- father GF P, parent P GC.
```

2.2.3. Introducing the existential quantifier in goals

Up to now the difference with Prolog is only superficial. We introduce progressively the new features of λProlog.

One may already use an explicit quantification by observing that variable P in *grandfather* does not occur in the head of the clause. Because of the intuitionistic equivalence $\forall x[(A\ x) \Rightarrow B] \equiv \exists x[(A\ x)] \Rightarrow B$, the quantification of P can be moved into the clause body and changed into an existential quantification, *sigma*.

```
grandfather GF GC :- sigma P \ (father GF P, parent P GC).
```

The program *grandfather* is so small that the explicit existential quantification does not really improve the expression. However, it contributes in general to an accurate classification of variables and of their scope (see Section 2.2.5). The need for such an accurate classification has already been recognized in Prolog in the context of either negation or the *setof* predicate [49].

2.2.4. Introducing implication in goals

We now assume the data-base also contains a record of "presumed fathers".

```
type presumed_father person -> person -> o.

presumed_father enoch methuselah.        ...
```

We want to combine the presumed father and grandfather relationship into the notion of "presumed grandfather" in which presumed fathers are regarded as fathers. There are two possibilities. First possibility, Prolog style, is to define relation *presumed_grandfather* on the model of relation *grandfather*.

```
type presumed_grandfather person -> person -> o.

presumed_grandfather PGF PGC :-
    (presumed_father PGF P; father PGF P),
    presumed_parent P PGC.
```

This is a bad idea because this definition ex nihilo does not show that relation *presumed_grandfather* contains relation *grandfather*. Moreover, the work of imitating an already existing relation is not limited to relation *grandfather*; one must also build a relation for *presumed-parent* which is almost a copy of predicate *parent*.

A second possibility is to reuse relation *grandfather* and assume in the definition of *presumed_grandfather* that relation *father* contains relation *presumed_father*. In doing so, the program structure displays the fact that relation *presumed_grandfather* contains relation *grandfather*. Indeed, intuitionistic logic is monotonic, and the structure of the program makes it easy to see that *presumed_grandfather* is simply *grandfather* with a supplementary axiom.

presumed_grandfather PGF PGC :–
((*pi F*\(*pi C*\(*father F C* :– *presumed_father F C*)))
=> *grandfather PGF PGC*).

This clause defines a presumed grandfather as an ordinary grandfather if one regards presumed fathers as fathers. The execution of a goal (*presumed_grandfather PGF PGC*) follows the deduction rule for intuitionistic implication: the clause $\forall$ *FC*[*father F C* $\Leftarrow$ *presumed_father F C*] is added to the program for the length of the proof of (*grandfather PGF PGC*). This is the intuitionistic interpretation of the implication. Contrary to the classical logic interpretation, one cannot give a trivial proof of an implication by merely disproving its premise.

Explicit quantifications of variables *F* and *C* at the assumption level (the two *pi*'s) are crucial. They play the same role as the quantifications that are implicit at the clause level in Prolog: they stand for $\forall_{\mathscr{D}}$, and they indicate that variables *F* and *C* must be renamed when solving a goal with the assumed clause. Because the default is to quantify at the outermost level, omitting their quantifications makes variables *F* and *C* free in the premise of the implication ('=>').

presumed_grandfather_2 PGF PGC :–
(*father F C* :– *presumed_father F C*
=> *grandfather PGF PGC*).

In doing so, they are not renamed when solving a goal with the assumed clause. This means that the assumed clause can only be used with one *F* and one *C*. In this application, it means the assumed clause can only be used once in a proof of *presumed_grandfather_2*. So, a *presumed_grandfather_2* is a *presumed_grandfather* with at most one presumptive link.

Clause assertion is Prolog's closest correspondent to clause implication. However, there are two major differences between the two. First, nothing forces Prolog to limit the life-time of an asserted clause to a subproof. Second, there cannot be any free variable in an asserted clause in Prolog. Indeed, the parameter of the built-in predicate that allows us to add a clause to a Prolog program (*assert*) is not itself a clause. It is a term that denotes a clause using the following rule: variables that are free in the term denote universally quantified variables of the asserted clause. For instance, executing *assert* (*p X*) in Prolog results in adding clause $\forall X$(*p X*), while executing (*p X*) $\Rightarrow$ *some_goal* in λProlog results in adding clause (*p X*) where *X* is left as a free existential variable in the added clause.

2.2.5. Introducing the universal quantifier in goals

In the previous examples, we limited ourselves to defining new relations using already defined relations. For instance, relation *grandfather* is defined as a join of

relations *father* and *parent*. Many other relations can be defined similarly. So, we want to abstract the process of defining relations as joins into a second-order relation. This relation could be used as follows:

```
grandfather GF GC :–
    join father parent GrandFather, GrandFather GF GC.
grandmother GM GC :–
    join mother parent GrandMother, GrandMother GM GC.
paternal_grandfather PGF GC :–
    join father father PatGrandFather, PatGrandFather PGF GC.
```

A possible definition of *join* is as follows:

```
type join (A–> B–> o) –> (B–> C–> o) –> (A–> C– > o) –> o.
% RJ is the join of R1 and R2
% ∀xy[R_J(x,y) ⇔ ∃J[R_1(x,J) ∧ R_2(J,y)]]
join R1 R2 RJ:–
    pi x \ (pi y\( (RJ x y) = (sigma J\( R1 x J, R2 J y )) )).
```

Note that it is logically higher-order (see Section 2.1.5) because *R1*, *R2* and *RJ* return type *o*. The universal quantifications specify that relation *RJ* must satisfy some intensional properties. They cannot specify extensional properties like that two relations have the same graph. For instance, $\forall x(parent\ x\ cain \Rightarrow parent\ x\ abel)$ is not provable. In other words, universal quantification in λProlog is not an enumeration over some domain. Even if in this example one may infer that x has type *person*, the proof of the universal goal will not try to replace it with every possible person (*adam*, *eve*, etc). Variable x is simply replaced by a new constant, distinct from every known person. If a proof is possible with this new constant, then it will be possible with every person. So, a goal $\forall x(G\ x)$ does not only say that every x (with the proper type) has the property G. It also says that there is a common proof for all the x's. This is the intensional interpretation of universal quantification.

In the definition above, we used a relation $=$ without declaring it. We could have declared it as an ordinary relation, plus an operator declaration for specifying its role as an infix operator, as follows:

```
type = A –> A –> o.
op 700 xfx =.

X = X.
```

We could also have built it locally in the definition of predicate *join*.

```
join R1 R2 RJ:–
  pi eq\(  pi X\(eq X X)
        =>  pi x\(pi y\(
              eq (RJ x y) (sigma J\(R1 x J, R2 J y)) )) ).
```

Here, the first *pi* is a universal quantification at the goal level, a $\forall_{\mathcal{G}}$. Together with the implication it models abstraction and modularity [33]. The second *pi* is a universal quantification at the clause level, a $\forall_{\mathcal{D}}$. It gives its scope to variable X.

In the example of relation *join*, and as opposed to relation *grandfather*, it is difficult to do without existential quantification. It avoids defining an intermediate predicate only for giving its proper scope to variable J. Without the existential quantification, the body of the clause would read

$$\exists J \forall xy[R_J(x,y) \Leftrightarrow (R_1(x,J) \wedge R_2(J,y))]$$

instead of

$$\forall xy[R_J(x,y) \Leftrightarrow \exists J[R_1(x,J) \wedge R_2(J,y)]].$$

2.2.6. Introducing λ-terms

Instead of giving a logical definition of what is a join, one may give a functional definition.

$$join\ R1\ R2\ x \setminus y \setminus (sigma\ J \setminus (R1\ x\ J, R2\ J\ y)).$$

Relation *join* displays some symmetry between universal quantification in goals and λ-abstraction. This symmetry is genuine even if it is neither absolute nor always as visible as in this example. The main theme of this article is to explore situations in which the symmetry exists, and to use it as a guide for programming.

3. A reconstruction of λProlog

We show how adding λ-terms to Prolog leads to all other λProlog component.

3.1. Adding λ-terms

3.1.1. Motivation

The motivation to manipulate λ-terms in logic programming is simple: they are the most natural representation for structures that combine either scoping, compositionality, or some form of genericity [32].

Logical and mathematical formulas feature scoping in quantifications (e.g., $\forall uP$, $\exists vP$), sums and products (e.g., $\sum_{x\in X} x$, $\prod_{i\in I} x_i$, $\int_0^1 f(x)\mathrm{d}x$), derivatives (e.g., $\mathrm{d}f/\mathrm{d}x$, $\partial f/\partial x$), etc, and computer programs feature scoping in parameterization (e.g., $f(x)$ *int* x; { ... }), blocks (e.g., { *int* x; ... }), etc. Expressions of compositional semantics feature compositionality as in denotational semantics (e.g., $\mathscr{T}_g[\![B_1, B_2]\!] = \lambda\kappa.(\mathscr{T}_g[\![B_1]\!]\ (\mathscr{T}_g[\![B_2]\!]\ \kappa))$ [50]) or Montague's semantics for natural language [45]. Finally, logical quantifications in automated deduction feature a form of genericity: one instantiates them using substitutions for building proofs.

The following table pictures some possible representations using λ-terms.

$\forall uP(u)$	*forall* $\lambda u(P\ u)$	$(=_\eta$ *forall* $P)$
$\sum_{x\in X} f(x)$	*sum* $X\ \lambda x(f\ x)$	$(=_\eta$ *sum* $Xf)$
$\int_0^1 f(x)dx$	*integral* 0 1 $\lambda x(f\ x)$	$(=_\eta$ *integral* 0 1 $f)$
df/dx	*derivative* $\lambda x(f\ x)$	$(=_\eta$ *derivative* $f)$
$f(x)$ *int* x; {...}	*function* f *int* $\lambda x(\ldots)$	
{*int* x; ...}	*block int* $\lambda x(\ldots)$	
$\mathscr{T}_g[\![B_1, B_2]\!] = \lambda\kappa.(\mathscr{T}_g[\![B_1]\!]$ $(\mathscr{T}_g[\![B_2]\!]\ \kappa))$	*t_g* (*B*1 *and B*2) λk (*D*1(*D*2 *k*)) ⇐ *t_g B*1 *D*1∧ *t_g B*2 *D*2	

Scoping introduces the notion of *scoped variable* or *parameter*. The key operation for composing structures is the *substitution* of a term for a parameter. Axiom β models such a substitution. This is why λ-terms are well suited for representing these structures: λ-abstraction serves as a generic quantification.

All these structures can be represented with first-order terms, but the correct handling of substitution with respect to scoping (e.g., avoiding capture of free variables) needs more care. The programmer must design among other things a representation for object-level variables. The most frequent solution is to represent them with Prolog variables (the *meta-variables*). This is called the *non-ground* representation. There are numerous examples of this solution in textbooks. They can be found in chapters dedicated to the manipulation of programs and formulas.

Consider as a very simple example the following clause in *The Art of Prolog* by Sterling and Shapiro [58] (program 3.29, p. 63): *derivative*(X,X,*s*(0)). It is part of a program that computes the derivative with respect to X of a function. One of its logical consequences is *derivative*(12,12,*s*(0)), which is absurd. Clause *polynomial*(X,X) (program 3.28, page 62) is a part of the definition of what is a polynomial in X. It has also absurd logical consequences: *polynomial*(12,12).

In Prolog, substitution of object-level variables is easy at the price of declarativity. It forces the programmer to check the correctness of object-level terms manipulations with respect to the operational semantics. Using λ-terms permits a coherent and declarative handling of scopes and substitutions.

Instead of these clauses, one could write the following clauses (and rewrite all the predicates accordingly):

derivative $x \setminus x \; x \setminus (s \; 0)$.

polynomial $x \setminus x$.

3.1.2. Adding $\alpha\beta$-equivalence and restricting to simple types

Once we admit that it is useful to integrate λ-calculus and logic programming one must decide how to do this so as to obtain the expected benefits and break nothing.

A first observation is that there is no use in adding λ-terms without a form of $\alpha\beta$-equivalence. Indeed, it is $\alpha\beta$-equivalence that allows us to substitute terms for variables with respect to scoping. For instance, the system

$$Prolog_\lambda \stackrel{\text{def}}{=} Prolog + \lambda\text{-}terms$$

without $\alpha\beta$-equivalence does not help much in manipulating scoped structures.

If λ-terms and a form of $\alpha\beta$-equivalence are added to Prolog, unification must be augmented to cope with axioms α and β. One wants also to be sure that there is always a head-normal form (see Section 2.1.3) to λ-terms in order to make the testing of equality feasible. This eliminates the pure λ-calculus but the domain of the simply typed λ-terms is eligible since it enjoys the strong normalization property. Another way of making the unification problem still less difficult is discussed in Section 5.2. There are other more sophisticated higher-order domains that also have the strong normalization property and a practical unification problem [13,54].

The unification problem for simply typed λ-terms modulo axioms α and β is semi-decidable and infinitary. The latter means that there can be infinitely many most general unifiers. For instance, the problem $\lambda u(N \; \lambda v(v) \; u) \stackrel{?}{=}_{\alpha\beta} \lambda w(w)$ has the following infinitely many minimal unifiers: $\theta_0 = [N \leftarrow \ulcorner 0 \urcorner], \theta_1 = [N \leftarrow \ulcorner 1 \urcorner], \; \theta_2 = [N \leftarrow \ulcorner 2 \urcorner], \ldots, \theta_i = [N \leftarrow \ulcorner i \urcorner]$, etc, where $\ulcorner . \urcorner$ denotes the Church encoding of natu-

ral numbers (see Section 2.1.3). The structure of the unification problem remains the same when one adds axiom η. However, the solutions may be different.

In practice, one uses the semi-algorithm proposed by Huet [26]. This semi-algorithm only searches for pre-unifiers, which are substitutions that make the problem trivially solvable, but are not necessarily solutions themselves. The difference between a pre-unifier and a unifier is that subproblems formed of flexible terms are considered solved and do not contribute to the solution. However, they are not discarded; they are simply added as constraints until they become more rigid. Huet's semi-algorithm is based on a possibly infinite search-tree that bears the pre-unifiers at its leaves. This search-tree is very similar to the search-tree that is used for searching proofs. Both trees are merged in practical implementations of λProlog, and proofs and multiple solutions are enumerated *via* the same mechanism, namely a depth-first traversal of the search-tree.

3.2. Structural induction on λ-terms

Programming by structural induction on a data-structure requires to be able to discriminate the various constructors of the data-structure, and to be able to access their components. Unification in Prolog does the discrimination and the accessing at the same time, and, due to its non-directional nature, it serves also for creating data-structures. λProlog's computation domain is the λ-terms, but we will see that it is impossible for λProlog's unification to discriminate between applications and λ-abstraction, and that it is impossible for unification alone to relate a given λ-abstraction and its body.

3.2.1. Discriminating applications from λ-abstractions in λProlog

A λ-term can be a constant, a variable, a λ-abstraction, or an application. It is impossible to discriminate λ-abstractions from applications, and to access their components using pure λProlog. Instead, one must use a specific programming discipline for representing object-level λ-abstractions and applications, and for manipulating them. The reason is that every λ-abstraction is β-equivalent to an application, and conversely for every application of function type.

One may compare these difficulties with well-known Prolog difficulties. The language of Prolog terms allows for variables in terms, but there is no means in pure Prolog for checking that a term is a variable. In fact, the closed model-theoretic semantics of Prolog actually uses a saturation of the term domain, the Herbrand universe, in which there is no room for variable terms. One needs an explicit notation of variables to manipulate them logically.

We intentionally use operation sounding words like "discriminate" and "access". This does not exclude a declarative reading of the problem. To distinguish between λ-abstractions and applications is to define a unary predicate that is true for every λ-abstraction and false for every application. Accessing applications requires to extend this predicate into a ternary predicate, *application*, that is true for every triple $\langle (A\ B), A, B\rangle$, and false for every other triple. Similarly, accessing λ-abstractions requires to define a binary predicate, *abstraction*, that is true of every pair $\langle \lambda x(E), E\rangle$, and false for every other combination of two terms.

The unification problem provides an intuition that a predicate like *application* cannot be defined. Indeed, terms 72, $(A\ B)$ and $\lambda x(x)$ can be unified with $(T_1\ T_2)$.

- Problem $72 \stackrel{?}{=}_{\alpha\beta} (T_1\ T_2)$ has two minimal solutions, $[T_1 \leftarrow \lambda x(x), T_2 \leftarrow 72]$ and $[T_1 \leftarrow \lambda x(72)]$.
- Problem $(A\ B) \stackrel{?}{=}_{\alpha\beta} (T_1\ T_2)$ has infinitely many minimal solutions, among them $[T_1 \leftarrow \lambda x(x\ B), T_2 \leftarrow\ A]$.
- Problem $\lambda x(x) \stackrel{?}{=}_{\alpha\beta} (T_1\ T_2)$ also has infinitely many minimal solutions, among them $[T_1 \leftarrow \lambda x(x), T_2 \leftarrow\ \lambda x(x)]$.

The solution for the discrimination problem is to label with a suitable constructor the applications and λ-abstractions we want to be able to recognize. For instance, the representation of object-level λ-terms in λProlog can use the following two constructors:

kind l_term type.
type app l_term –> l_term –> l_term.
type abs (l_term–> l_term) –> l_term.

Using constructors *app* and *abs*, one can encode every closed term of the pure untyped λ-calculus. One must write $(app\ F\ X)$ instead of $(F\ X)$ and $(abs\ x \setminus (app\ (app\ +\ x)\ 12))$ instead of $x \setminus (x + 12)$. Using this technique, a distinction is introduced between object-level and meta-level λ-terms, and one can discriminate λ-abstractions from applications at the object-level. One can also access the components of object-level applications as follows:

type application l_term → l_term → l_term → o.
application (app A B) A B.

This encoding does not model all aspects of the semantics of the object-level λ-terms. For instance, it is not true that $(app\ (abs\ E)\ F) =_{\alpha\beta} (E\ F)$. If such a relation exists at the object level, it must be described explicitly in λProlog in an *ad hoc* relation: for instance,

type beta_conv l_term –> l_term –> o.
beta_conv (app (abs E) F) (E F).

Term F is substituted for the variable bound in $(abs\ E)$ by the meta-level β-reduction of $(E\ F)$

One may wonder why there is no constructor for λ-variables. In fact, it is useless and it would make the manipulation of object λ-terms even more indirect. A constructor for λ-variables would be declared as

type var l_term –> l_term.

such that $x \setminus x$ is represented as $(abs\ x \setminus (var\ x))$. Such an encoding is useless because as soon as λ-abstractions are discriminated by a constructor, say $(abs\ E)$, all occurrences of a bound variable are accessible *via* β-reduction, $(E\ something)$. The discipline presented in the sequel shows that whenever λ-variables must be recognized for some purpose, there is a way to distinguish them without using constructor *var* (see predicate *nnf* in Section 3.4 and predicate *de_bruijn* in Section 3.5). Furthermore, a constructor like *var* makes the manipulation of object λ-terms more indirect because β-reduction would result in terms with idle *var* constructors in them: e.g., $(x \setminus (var\ x)\ F) =_{\alpha\beta} (var\ F) \neq_{\alpha\beta} F$. These unnecessary *var's* must be tackled for in the implementation of the object-level manipulations. In fact, *var* is essentially identity, but one needs to implement an explicit rewrite rule for it.

The problem of accessing the body of an object-level λ-abstraction is solved in the next section. Indeed, the technique proposed for discriminating λ-abstractions does not tell what to substitute to ?? in the following program:

type abstraction l_term → l_term → o.
abstraction (abs E) ?? .

3.2.2. Relating λ-abstractions and their bodies

It is important to observe that there can be no free λ-variable in an argument of a predicate. Indeed, renaming *via* α-equivalence concerns only bound variables. If one could prove (*abstraction* $\lambda x(x)$ x), then using α-equivalence (*abstraction* $\lambda y(y)$ x) would also be provable. So, (*abstraction* A C) can be a consequence of some program only if there is no free λ-variable in C. In other words, A is a λ-abstraction $\lambda x(C)$ where x does not occur free in C. So, the best that a predicate like *abstraction* can do is to discriminate the λ-terms that are $\alpha\beta$-equivalent to a constant function.

It is easy to check that the syntax of λProlog does not permit free λ-variables in the arguments of a goal; non-bound symbols are either names of constants in $\mathscr{C}_t$, or names of variables bound by implicit $\forall_{\mathscr{D}}$'s. One can also show that computation keeps this property true of binding values of free existential variables and of arguments of goals [7]; they cannot contain free occurrence of λ-variables.

The impossibility of having free occurrences of λ-variables in the goals that are consequences of a program can be phrased as follows: the computation domain of λProlog, its "Herbrand universe", consists of the *combinators* (see Section 2.1.2) of the simply typed λ-calculus, and not just any simply typed λ-terms.

The principal consequence of this observation is that there is no direct means to access the body of a λ-abstraction. Indeed, it generally contains free occurrences of the λ-variable that the λ-abstraction binds. An indirect means for accessing the body of a λ-abstraction A without capturing the λ-variable it binds, is to apply A to a term t and use term $A' = (A\ t)$ instead of A. This "consumes" one λ-abstraction and all the occurrences of the bound λ-variable.

For the sake of reversibility of computation, given A' and t, one must be able to compute A by solving $A' \stackrel{?}{=}_{\alpha\beta} (X\ t)$ for an unknown X. So, the equality theory of the λ-terms and the term t must be such that given two non-equivalent terms A and B, the terms $(A\ t)$ and $(B\ t)$ are not equivalent. We call this condition the *conservation condition*. It ensures that accessing the body of a λ-abstraction is a reversible operation. As a counter-example, consider $A = \lambda x(x)$, $B = \lambda x(1729)$, and $t = 1729$; A and B are not $\alpha\beta$-equivalent though $(A\ t) =_{\alpha\beta} (B\ t)$.

We will show that the conservation condition is satisfied if and only if the following two conditions hold:

1. the equality theory of λ-terms entails η-equivalence,
2. the term t is recognizable (in a sense that will be made clear).

3.3. The need for axiom η

To apply a λ-abstraction to a term in order to access its body does not allow us to discriminate between η-equivalent terms (e.g., E and $\lambda x(E\ x)$). Indeed, for all t, $(\lambda x(E\ x)t) =_{\alpha\beta} (E\ t)$ even if $\lambda x(E\ x) \neq_{\alpha\beta} E$. Hence, terms E and $\lambda x(E\ x)$ are two non-$\alpha\beta$-equivalent terms to which always correspond two $\alpha\beta$-equivalent terms when they are applied to some term. Such pair of terms, considered under $\alpha\beta$-equivalence, always violates the conservation condition. So, the system

$$\alpha\beta Prolog \stackrel{\text{def}}{=} Prolog + \lambda\text{-}terms + simple\ types + =_{\alpha\beta} + \cdots$$

(i.e., λProlog without η-equivalence) is not a safe logic system for programming by induction on the structure of λ-terms. In this system, one could conclude that two structures are equivalent, though they are not.

To satisfy the conservation condition, η-equivalence must be added to the equality theory. In doing so, terms E and $\lambda x(E\ x)$ are considered as equivalent, and the fact that, for all t, $(\lambda x(E\ x)\ t)\ =_{\alpha\beta}\ (E\ t)$ is no longer a problem.

3.4. The need for universal quantification in goals: $\forall_{\mathscr{G}}$

We have shown how we plan to analyze a λ-abstraction A by applying it to some term t and analyzing the result $A' = (A\ t)$. If one wants to be able to compute A by solving $A' \stackrel{?}{=}_{\alpha\beta}(X\ t)$ for an unknown X, one needs more than η-equivalence. Informally, t must be recognizable among the subterms of A'.

3.4.1. Discussion

Let us give an example of what happens when t is not recognizable. Let $A = \lambda x(x + 12)$ and $t = 12$, we get $A' = (12 + 12)$. The problem $(12 + 12) \stackrel{?}{=}_{\alpha\beta\eta}(X\ 12)$ has four solutions with no formal reason to prefer one: $[X \leftarrow\ \lambda x(x + x)]$, $\underline{[X \leftarrow \lambda x(x + 12)]}$, $[X \leftarrow\ \lambda x(12 + x)]$, and $[X \leftarrow\ \lambda x(12 + 12)]$. They correspond to four A's that are not $\alpha\beta\eta$-equivalent but still yield $\alpha\beta\eta$-equivalent $(A\ t)$'s for $t = 12$. Given our objective of reconstructing A, the underlined solution is the only acceptable one. We need some formal means to select it.

One could object that it was not really clever to choose $t = 12$ when A already had 12 as a subterm. This objection does not hold for two reasons. First, because a term A can have unknown subterms, it is not always possible to check that a candidate t does not occur in A. Second, even when the term A is fully determined (ground, see Section 2.1.5), to choose a t that does not occur in it does not completely solve the problem. For instance, if $A = \lambda x(x + 12)$ and $t = 13$, we get $A' = (13 + 12)$. Problem $(13 + 12) \stackrel{?}{=}_{\alpha\beta\eta}(X\ 13)$ has two solutions among which there is still no formal means for selecting one: $\underline{[X \leftarrow \lambda x(x + 12)]}$ and $[X \leftarrow\ \lambda x(13 + 12)]$. Again, two non-$\lambda$-equivalent A's yield two λ-equivalent $(A\ t)$'s for $t = 13$.

In fact, for any problem $A' \stackrel{?}{=}_{\alpha\beta}(X\ t)$, where t is an ordinary term and $A' = (A\ t)$, there are always at least two solutions in X: $[X \leftarrow\ \lambda x(A\ t)]$ and $\underline{[X \leftarrow A]}$. The term t must be special enough to eliminate exactly all the solutions that contain it.

Universal quantification in goals, $\forall_{\mathscr{G}}$, is a logical means for producing a recognizable term t. The universal constant that is introduced by the deduction rule for $\forall_{\mathscr{G}}$ has exactly the desired property (see Section 2.1.5). Indeed, if t is universally quantified in the scope of the quantifications of A and X, then the only solution of equation $(A\ t) \stackrel{?}{=}_{\alpha\beta\eta}(X\ t)$ is $\underline{[X \leftarrow A]}$. More precisely, the goal to be proved has the following form: $\exists A \exists X \forall t[(A\ t) = (X\ t)\ ]$. Because of η-equivalence, this goal is equivalent to $\exists A \exists X[\ A = X\ ]$, which has the trivial solution $\underline{[X \leftarrow A]}$.

This shows a fundamental correspondence between λ-abstraction and universal quantification via η-equivalence. They can be used respectively in terms and formulas for reflecting each other. This is what we call the *symmetry* between λ-abstraction and universal quantification.

3.4.2. Example

Let us define a predicate that relates a first-order predicate calculus formula and its negation normal form [16]. A first-order predicate calculus formula is in negation normal form if the negation connective is only applied to atomic formulas. For instance, $(\neg A) \vee (\neg B)$ is in negation normal form, but $\neg(A \wedge B)$ is not. It is always possible to transform a first-order predicate calculus formula into an equivalent negation normal formula using De Morgan's identities.

We need two types for representing the formulas of the first-order predicate calculus: a type for formulas, and a type for individuals. They are *object-level* formulas and individuals. They are represented by λProlog λ-terms, but they must not be mistaken for λProlog formulas or λ-terms.

kind (*formula, individual*) *type*.

One also needs connectives for object-level formulas,

type (*and, or*) *formula –> formula –> formula*.

type not formula –> formula.

type (*forall, exists*) (*individual–> formula*) *–> formula*.

and object-level propositional constants,

type $(p, \ldots)$ *individual –> individual –> formula*.

type $(q, \ldots)$ *formula*.

So, formula $\forall x[p(x,x) \vee q]$ is encoded by the λ-term (*forall x\(or* (*p x x*) *q*)).

The only constructors that are original with respect to Prolog are *forall* and *exists*. They have an argument that is a λ-abstraction of λProlog. Its role is to formalize the scope of object-level quantifications and to handle the fact that quantified variables can be substituted (e.g., in proofs).

The semantics of object-level connectives must be defined by deduction rules, axioms, etc., written in λProlog. We describe by structural induction the relation between a first-order predicate calculus formula and its negation normal form. The cases for all the connectives are elementary and could be described in Prolog.

```
type nnf formula –> formula –> o.

nnf (and F1 F2) (and G1 G2) :– nnf F1 G1, nnf F2 G2.
nnf (or F1 F2)  (or G1 G2) :– nnf F1 G1, nnf F2 G2.
nnf (not (and F1 F2))  (or G1 G2) :– nnf (not F1) G1, nnf (not F2) G2.
nnf (not (or F1 F2)) (and G1 G2) :– nnf (not F1) G1, nnf (not F2) G2.
nnf (p A B) (p A B).
nnf (not (p A B)) (not (p A B)).
...
```

The clauses for quantifications are more interesting because one must continue the structural induction in the quantified formula. A universal quantification of λProlog(*pi*) is required for going through the λ-abstraction.

```
nnf (forall F) (forall G) :– pi i\ (nnf (F i) (G i)).
nnf (exists F) (exists G) :– pi i \ (nnf (F i) (G i)).
nnf (not (forall F)) (exists G) :– pi i \ (nnf (not (F i)) (G i)).
nnf (not (exists F)) (forall G) :– pi i \ (nnf (not (F i)) (G i)).
```

In relation *nnf*, we use λProlog universal quantification in a way that has nothing to do with the semantics of object-level formulas. It has only to do with their

structure. The two object-level quantifications, *exists* and *forall*, respectively existential and universal, are both handled by a universal quantification of λProlog, and it is used only because the parameter of exists and *forall* is of functional type.

When we use predicate *nnf* for normalizing a formula, the universal quantification analyzes and synthesizes λ-abstractions at the same time. The λ-abstraction F is analyzed and its body passed as an argument to the recursive call to *nnf*. The second argument is unified with a term (*exists* G) or (*forall* G) where G is an unknown. The application of G to i is also passed as an argument to the recursive call to *nnf*. The unknown G being quantified out of the scope of the i cannot have any occurrence of i in its binding values. Hence, it will be bound to a λ-abstraction that will become more and more precise as long as formula F is traversed.

We present the processing of an actual proof for illustrating the last observation.

1. The query is (*nnf* (*exists* $x \setminus$ (*not* (*exists* $y \setminus (p\ x\ y)$))) X).
2. Resolution with the clause for *exists*-formulas produces substitution $[X \leftarrow (exists\ G)]$, and the goal *pi* $i \setminus$ (*nnf* (*not* (*exists* $y \setminus (p\ i\ y)$)) $(G\ i)$).
3. Rule $\forall_{\mathcal{G}}$ (Section 2.1.5) produces goal (*nnf* (*not* (*exists* $y \setminus (p\ c\ y)$))$(G\ c)$).
4. Resolution with the clause for *not-exists*-formulas leads to solving problem $(G\ c) = (forall\ G')$. Its unique most general solution is $[G \leftarrow x \setminus (forall\ (H\ x)), G' \leftarrow (H\ c)]$ [26]. The new goal is *pi* $i \setminus$(*nnf*(*not*$(p\ c\ i)$)$(H\ c\ i)$).
5. Rule $\forall_{\mathcal{G}}$ produces goal (*nnf*(*not*$(p\ c\ c')$)$(H\ c\ c')$).
6. Resolution with the clause for *not-p*-formulas produces substitution $[H \leftarrow x \setminus y \setminus (not\ (p\ x\ y))]$, and goal *Success*.
7. The solution is $[X \leftarrow (exists\ x \setminus (forall\ y \setminus (not(p\ x\ y))))]$.

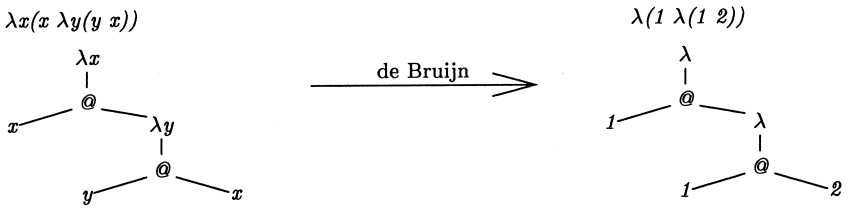
3.5. The need for implication in goals: $\Rightarrow_{\mathcal{G}}$

3.5.1. Discussion and example

We have seen how the universal quantification allows us to handle λ-abstractions. We also have seen in Section 2 that the deduction system interprets universal quantifications by substituting a *new* constant for the universal variable. The constant is simply added to the current signature (Section 2.1.5).

This new constant is a problem: how can the programmer take it into account? Because the constant is new, no predicate definition can take it into account in the initial program. However, it belongs to some type, and predicates that are supposed to be defined for every constructor of this type are not defined for the new constant. The fact that all constructors of a type must be declared gives an impression of closedness of the type, but universal quantification brings in a form of openness: new constructors can be added at any time.

We illustrate the problem with the definition of a predicate that relates λ-terms and their de Bruijn's notation [12]. Hannan presents a similar example in the framework of extended natural semantics [20]. The purpose of de Bruijn's notation is to avoid, by simply eliminating names, the naming problem that is otherwise solved using axiom α. The idea is to replace every occurrence of a bound name by the number of λ-abstractions that are between this occurrence and the λ-abstraction that binds the name; including this λ-abstraction. The following picture illustrates de Bruijn's notation in textual and graphical representations.



Object-level λ-terms are represented using constants *app* and *abs* (see Section 3.2.1). De Bruijn trees are represented by the following constants:

kind db_tree type.
type db_app db_tree –> db_tree –> db_tree.
type db_abs db_tree –> db_tree.
type db_var int –> db_tree.

We assume that a type *int* is defined with constructors *succ* and *zero* and a relation *plus*: (*plus A B X*) if and only if $A + B = X$. Hence, the λ-term of the example that illustrates de Bruijn's notation is written as

(*abs x \ (app x (abs y \ (app y x))))*,

and its de Bruijn tree is written as

```
(db_abs (db_app (db_var (succ zero))
     (db_abs (db_app (db_var (succ zero))
          (db_var (succ (succ zero)))))))).
```

Let us describe in λProlog the relationship between a variable-free object-level term, its de Bruijn tree, and the number of λ-abstractions inside which the term occurs. Defining relation *de_bruijn* by structural induction starts as follows:

```
type de_bruijn l_term –> db_tree –> int –> o.

de_bruijn (app T1 T2) (db_app D1 D2) N :–
   de_bruijn T1 D1 N, de_bruijn T2 D2 N.
de_bruijn (abs T) (db_abs D) N:– pi x\(de_bruijn (T x) D (succ N)).
```

Executing this predicate, one may use the second clause, and eventually reach the universal variable x at some leaf of the term. At this point, one would like to use a clause like

de_bruijn x (db_var I_x) N_x:– plus N I_x N_x.

where the x is the universal variable introduced above, N_x is the depth at which this occurrence of x occurs, and N is the depth at which x is bound. So, x and N should belong to the context of the clause that introduced x. This contradicts the convention that the scope of a variable identifier is limited to the clause in which it occurs. Variables N_x and I_x are normal and are local to the clause. Variable N_x takes its value from the calling goal, and I_x takes it from the clause body.

One needs such a clause for every x introduced when predicate *de_bruijn* is executed. This is not possible in a framework in which the program is a static entity, because one does not even know how many x there may be. We need some means

for augmenting a predicate definition during the life of every x: i.e., during a subproof. Moreover, this mechanism must allow us to add clauses that have free variables in them (like x and N above). Implication in goals, $\Rightarrow_{\mathcal{G}}$, has these properties.

With implication, one can write the clauses for *abs* and all x's as follows:

```
de_bruijn (abs T) (db_abs D) N :-
  pi x\ (   pi N_x\ (
  pi I_x\ (de_bruijn x (db_var I_x) N_x:- plus N I_x N_x))
     =>   de_bruijn (T x) D (succ N)).
```

The *pi*'s in the implied clause make variables N_x and I_x local to the implied clause. They correspond to the universal quantifications that are usually implicit at the clause level. In this case, the nesting of clauses forces us to make them explicit to avoid a confusion with the outermost clause. Variable N is intentionally left free in the implied clause. Hence, it is shared by the two clauses. The universal variable x is shared by the premise and the conclusion of the implication.

An implication-less solution to this problem that does not introduce a *var*-like constructor (see Section 3.2.1) is to accumulate the dynamic clauses in a list. Since the accumulated clauses are not meant to be directly executed, only their relevant parts are accumulated.

```
type de_bruijn l_term -> db_tree -> int -> (list o) -> o.
type db_x l_term -> int -> o.

de_bruijn (app T1 T2) (db_app D1 D2) N L :-
  de_bruijn T1 D1 N L, de_bruijn T2 D2 N L.
de_bruijn (abs T) (db_abs D) N L :-
  pi x\(de_bruijn (T x) D (succ N) [db_x x N / L]).
de_bruijn X (db_var I_x) N_x L :-
  member (db_x x N) L, plus N I_x N_x.
```

This solution works as well as with implication, but we think it is less readable because the logic of the program comes from the logic of the list manipulation, which must be deciphered, whereas, when using implication, the logic of the program comes from the logic of its connectives.

There are however legitimate circumstances in which one must avoid implication. This is when the logic of implication is not exactly the desired logic. Indeed, λProlog-implication (i.e., intuitionistic implication) ensures neither a determined selection order, nor that all assumed clauses will be used, nor that they will be used only once. All these properties may be necessary in applications like natural language processing. Note that linear logic [18] models these properties, and that some fragments and some presentations of it have the uniform proof property [23,31].

3.5.2. Another example [20]

Let us define a predicate that relates a pure λ-calculus term to its simple type. One needs a type for representing the object-level simply typed λ-calculus, and a type for the object-level simple types. For object-level λ-terms, we will reuse constants *l_term*, *app* and *abs* (see Section 3.2.1). One also need new constructors for representing simple types, and various term and type constants of the object-level.

kind s_type type.
type arrow s_type –> s_type –> s_type.
type (*integer, real, ...*) *s_type.*
type (*zero, successor, ...*) *l_term.*

Then, the well-typing relation is defined by structural induction on the constructors of type *l_term*. The case for elementary constructors and applications is elementary and could be written in Prolog.

type typing l_term –> s_type –> o.

typing zero integer.
typing successor (*arrow integer integer*).
...
typing (*app T1 T2*) *B* :– *typing T1* (*arrow A B*), *typing T2 A.*

The case for λ-abstractions is more interesting. The logic of the well-typing relation is given by the deduction rules of the theory of simple types [2,55]. For the λ-abstraction, the rule is called *arrow introduction*. Note that a typing sequent $\Gamma \vdash t : \tau$ reads "term t has type τ in context Γ".

$$\frac{\Gamma, x{:}\,\alpha \vdash E{:}\,\beta}{\Gamma \vdash \lambda x(E){:}\,\alpha \to \beta} \to_I$$

When interpreted operationally (i.e., bottom-up), it shows that the structural induction must continue through the λ-abstraction. Structural induction through object-level λ-abstractions uses a universal quantification because object-level λ-abstractions are represented by λProlog λ-abstractions (see Section 3.4). This universal quantification introduces a new constant of type *l_term*, which is the type on which the structural induction operates. So, one must augment the definition for the life-time of the new constant using an implication (see discussion and first example of this section)

typing(*abs E*)(*arrow A B*) *g*:– *pi x* \ (*typing x A* => *typing* (*E x*)*B*).

The added clause is (*typing x A*). It contains a free existential variable *A*. This forces all the occurrences of constant x in the normal form of (*E x*) to be assigned the same object-level type by predicate *typing*. If the added clause had been $\forall T$(*typing x T*), every occurrence would have had its own object-level type.

4. Programming by structural induction

The practical advantage of our reconstruction of λProlog is that it provides a guide for programming. The first thing to do for programming is to define data-structures (i.e., type and term constants, $\mathscr{K}_i$ and $\mathscr{C}_t$ of Sections 2.1.1 and 2.1.2) for representing object-level structures. For instance, (*list T*) is a data-structure type, and *nil* and *cons* are its constructors. Then, relations on the object-level structures can be defined via structural induction on the λ-terms of the meta-level.

4.1. Inductiveness

We recall that according to the *Curry–Howard isomorphism* [25] the arrow of simple types is analogous to the implication of the propositional intuitionistic calculus.

As does implication, the arrow introduces a notion of positive and negative occurrences as follows:

$$
\begin{array}{lcll}
pos(A \rightarrow B) & \stackrel{\text{def}}{=} & neg(A) \cup pos(B) & \\
neg(A \rightarrow B) & \stackrel{\text{def}}{=} & pos(A) \cup neg(B) & \\
pos(T) & \stackrel{\text{def}}{=} & \{T\} & \textit{if } T \textit{ is not an arrow type} \\
neg(T) & \stackrel{\text{def}}{=} & \emptyset & \textit{if } T \textit{ is not an arrow type}
\end{array}
$$

For instance, $pos((a \rightarrow b) \rightarrow (c \rightarrow d)) = \{a, d\}$ and $neg((a \rightarrow b) \rightarrow (c \rightarrow d)) = \{b, c\}$.

One says that a data-structure type ϕ is *inductive* if all its constructors' parameters have types in which ϕ has only positive occurrences [55]. The notion of inductiveness extends easily to data-structures defined by mutual recursion. By extension, we call inductive a constructor of a data-structure type ϕ if all its parameters have types in which ϕ has only positive occurrences. We call a constructor non-inductive in the opposite case: i.e., at least one of its parameters has a type in which ϕ has a negative occurrence.

Inductiveness gives us a rationale for using implication goals without having to recourse to operational reasoning as we did for predicates *de_bruijn* and *typing* (see Section 3.5). Negative occurrences correspond to occurrences of the universal variables that interpret λ-abstraction. If a data-structure type is inductive then it is easy to deduce an induction scheme on its constructors [4,55]. If it is not, there are negative occurrences of the type being defined, and one needs to extend the induction function at run time using implication.

Let us consider the examples of the previous sections. The definition of relation *nnf* does not use implication. This is because the data-structure type *formula* is inductive. Let us show that the types of the parameters of its constructors contain the type *formula* in positive occurrences only. We observe that the constructors are either like *and*, or like *forall*. Every parameter of *and* has type *formula*, and the only parameter of *forall* has type $individual \rightarrow formula$. We have $neg(formula) = \emptyset$ and $neg(individual \rightarrow formula) = \{individual\}$. So, *formula* is an inductive type because it does not occur in negative occurrences in the types of its constructors' parameters.

Symmetrically, the two examples on λ-terms use implication. Let us show that the data-structure type *l_term* is not inductive. Its constructors are *app* and *abs*, and the type of *abs* is $(l\_term \rightarrow l\_term) \rightarrow l\_term$. The type of the only parameter of *abs* is $l\_term \rightarrow l\_term$. We have $neg(l\_term \rightarrow l\_term) = \{l\_term\}$. Type *l_term* has a negative occurrence in the type of a parameter of one of its constructors; therefore it is not inductive.

In his work on representing higher-order unification problems as logic programs [38], Miller also builds a program by structural induction on types, but the polarities of the types do not matter and implications are simply generated for every arrow type. In his thesis [29], Liang adds polarities to this scheme. However, this is not for deciding whether an implication is to be used or not, but for deciding what to imply. In fact, their goal is not the same as ours; we define a relation for a given argument type, whereas they define a generic program that should work for every type. So, they consider a simultaneous mutually recursive definition of all the types of a signature, which makes every negative occurrence non-inductive.

4.2. Higher-orderness

The notion of the functional order of a type can be formally defined as follows:

$$ord(A \to B) \stackrel{\text{def}}{=} max(ord(A)+1, ord(B))$$

$$ord(T) \stackrel{\text{def}}{=} 0 \quad \textit{if } T \textit{ is not an arrow type}$$

For instance, $ord((a{\to}b){\to}(c{\to}d)) = 2$, $ord(a{\to}a) = ord(a{\to}a{\to}a{\to}a) = 1$, and $ord(a) = 0$. Note that a type of order 0 or 1 is always inductive. A type of order strictly greater than 1 is called *higher-order*. By extension, we say that a term (in particular, a constant) has the order of its type.

Given the extended definitions of inductiveness and higher-orderness, any constructor $c_i \in \mathscr{C}_t$ of a data-structure type ϕ (i.e., t is $\tau_1 \to \ldots \to \tau_n \to \phi$) can be classified as first-order (i.e., $ord(t) \leqslant 1$), higher-order and inductive (i.e., $ord(t) > 1$ and $\forall i[\phi \notin neg(\tau_i)]$), or higher-order and not inductive (i.e., $ord(t) > 1$ and $\exists i[\phi \in neg(\tau_i)]$). For instance, constant *app* (see Section 3.2.1) is first-order, *forall* (see Section 3.4.2) is higher-order (in fact order 2) inductive, and *abs* (see Section 3.2.1) is higher-order (also order 2) non-inductive.

4.3. Structural induction

We propose a general scheme for defining relations by structural induction on λ-terms. The ultimate goal of such a formalization is to provide a semi-automated environment for programming in the structural induction style.

Let a relation R with type $\phi \to \psi \to o$ be defined by structural induction on its first argument. We call the first argument the "induction" argument, and its type (ϕ) the "induction type". We call the second argument the "output" by analogy with the functional case. We assume that all constructors of the induction type are declared so that they can be classified as first-order, higher-order inductive, and higher-order non-inductive. A clause is associated to every constructor c_i. It is built after a pattern that depends only on the constructor and is parameterized by a relation R_{c_i} that depends on the constructor and the relation to be defined.

4.3.1. First-order case

The clause pattern associated to a first-order constructor c_i of type $\ldots \to \phi$ is as follows:

$$R\ (c_i\ t_1 \cdots\ t_{a_i})\ O \Leftarrow$$
$$R\ t_{\rho_1}\ O_{\rho_1} \wedge \cdots \wedge R\ t_{\rho_{r_i}}\ O_{\rho_{r_i}} \wedge\ R_{c_i}\ t_1 \cdots\ t_{a_i}\ O_{\rho_1} \cdots\ O_{\rho_{r_i}}\ O$$

where a_i is the arity of c_i, r_i is the number of arguments of c_i that have type ϕ, and $\rho_1, \ldots, \rho_{r_i}$ are their ranks.

The clauses are entirely determined by ϕ (and its constructors) and by the R_{c_i}'s. Given ϕ, the clause patterns can be produced automatically. A relation of fixed type, $\phi \to \psi \to o$, definable via structural induction on ϕ, is thus fully determined by the R_{c_i}'s.

For instance, given the type *list* (see Section 2) the clause patterns are:

$$R\ \text{nil}\ O \Leftarrow R_{nil}\ O$$
$$R\ (\text{cons}\ t_1\ t_2)\ O \Leftarrow R\ t_2\ O_2 \wedge R_{cons}\ t_1\ t_2\ O_2\ O$$

and the relation that associates a list and its length is defined by

$R_{nil} = \lambda o(o = zero)$
$R_{cons} = \lambda t_1 \lambda t_2 \lambda o_2 \lambda o(o = (succ\ o_2))$
These clause patterns and relations determine the following predicate:
r [ ] *O* :– *O* = *zero*.
r [*T1* / *T2*] *O* :– *r T2 O2*, *O* = (*succ O2*).
It can be simplified for producing the usual program.

4.3.2. Higher-order inductive case

The problem of structural induction at higher-order is that a subterm of a given type is not necessarily built with constructors of this type; it can be built with a λ-variable introduced by a constructor of higher-order type. There is no difficulty if no occurrence of a λ-variable can return a value of the induction type, and this is precisely what inductiveness ensures. The rule pattern for higher-order inductive constructors has the following form.

$R\ (c_i\ t_1 \cdots\ t_{a_i})\ O\ \Leftarrow$
$\quad \hat{R}\ t_{\rho_1}\ O_{\rho_1} \wedge \cdots \wedge \hat{R}\ t_{\rho_{r_i}}\ O_{\rho_{r_i}} \wedge R_{c_i}\ t_1 \cdots t_{a_i}\ O_{\rho_1} \cdots\ O_{\rho_{r_i}}\ O$

where a_i is the arity of c_i, r_i is the number of arguments of c_i which have type ϕ or $\ldots \to \phi$, $\rho_1, \ldots, \rho_{r_i}$ are their ranks, and $\hat{R}\ t_{\rho_i}\ O_{\rho_i}$ is either $R\ t_{\rho_i}\ O_{\rho_i}$ if $typeof(t_{\rho_i}) = \phi$ or $\forall \overline{x}[\ R\ (t_{\rho_i}\ \overline{x})\ (O_{\rho_i}\ \overline{x'})\ ]$ otherwise, where the number and types of the $\overline{x}$ correspond to the arguments expected by t_{ρ_i}, and the $\overline{x'}$ are a subset of the $\overline{x}$ according to the type of the output.

The $\forall$'s in the second case for $\hat{R}\ t_{\rho_i}\ O_{\rho_i}$ will be written *pi* in concrete λProlog programs.

4.3.3. Higher-order non-inductive case

In the non-inductive higher-order case, a universal constant may correspond to a negative occurrence of the type of interest. Every negative occurrence of ϕ in the type of a parameter of one of its constructors is considered as a family of constructors for ϕ. We associate to such a family of constructors a relation $S_{c_i,j}$ that is similar to relations R_{c_i} except for a supplementary parameter for handling the context in which the constructor has been introduced. The context is also passed to the relation associated to the non-inductive constructor. The rule pattern for higher-order non-inductive term constructors has the following form.

$R\ (c_i\ t_1 \cdots\ t_{a_i})\ O\ \Leftarrow$
$\quad \hat{R}\ C\ t_{\rho_1}\ O_{\rho_1} \wedge \cdots \wedge \hat{R}\ C\ t_{\rho_{r_i}}\ O_{\rho_{r_i}} \wedge R_{c_i}\ C\ t_1 \cdots t_{a_i}\ O_{\rho_1} \cdots\ O_{\rho_{r_i}}\ O$

where a_i is the arity of c_i, r_i is the number of arguments of c_i that have type ϕ or $\ldots \to \phi$, $\rho_1, \ldots, \rho_{r_i}$ are their ranks, C is an existential variable that represents the context, and $\hat{R}\ C\ t_{\rho_i}\ O_{\rho_i}$ is either $R\ t_{\rho_i}\ O_{\rho_i}$ if $typeof(t_{\rho_i}) = \phi$, or $\forall \overline{x}[\ R\ (t_{\rho_i}\ \overline{x})\ (O_{\rho_i}\ \overline{x'})\ ]$ if $typeof(t_{\rho_i}) = Argtypes \to \phi$ and ϕ has only positive occurrences in the *Argtypes*, or

$\forall \overline{x}[\forall o_{v_1}[R\ x_{v_1}\ o_{v_1} \Leftarrow\ S_{c_i,v_1} C\ x_{v_1} o_{v_1}]$
$\quad \Rightarrow \ldots$
$\quad \Rightarrow \forall o_{v_{n_i}}[R\ x_{v_{n_i}}\ o_{v_{n_i}} \Leftarrow\ S_{c_i,v_{n_i}}\ C\ x_{v_{n_i}} o_{v_{n_i}}] \Rightarrow\ R\ (t_{\rho_i}\ \overline{x})\ (O_{\rho_i}\ \overline{x'})\ ]$

otherwise, where n_i is the number of negative occurrences in the *Argtypes* and $v_1, \ldots, v_{n_i}$ are their ranks, $\overline{x}$ correspond to the arguments expected by t_{ρ_i}, and $\overline{x'}$ is a subset of $\overline{x}$ according to the type of the output.

The $\forall$'s in the second and third cases for $\hat{R}\ t_{\rho_i}\ O_{\rho_i}$ will be written *pi* in concrete λProlog programs, and the $\Rightarrow$ in the third case will be written =>.

For instance, given type *l_term*, the typing relation is defined by three relations
$R_{app} = \lambda t_1 \lambda t_2 \lambda o_1 \lambda o_2 \lambda o(o_1 = (arrow\ o_2\ o))$
$R_{abs} = \lambda c \lambda t_1 \lambda o_1 \lambda o(o = (arrow\ c\ o_1))$
$S_{abs,1} = \lambda c \lambda x \lambda o(o = c)$
which determine a predicate

```
r (app T1 T2) O :- r T1 O1, r T2 O2, O1 = (arrow O2 O).
r (abs T1) O :-
   pi x\(pi Ox\(r x Ox :- Ox = C) => r (T1 x) O1),
   O = (arrow C O1).
```

which can be simplified into predicate *typing* (see Section 3.5.2).

5. Discussion

We summarize our reconstruction of λProlog, and we give links to related topics.

5.1. The structure of λProlog

Our reconstruction considers λ-terms and αβ-equivalence as the principal components of λProlog, those that make every other component necessary. Adding these components to Prolog is motivated by the need for a more declarative handling of notions such as scoping and substitution.

The λ-calculus used for this purpose must be restricted in order to make unification well-defined and tractable. Simple types produce the desired restriction together with an ML-like typing discipline. The computation domain of λProlog (its "Herbrand universe") must be restricted to combinators to make α-equivalence compatible with logical deduction. So, there is no direct means for manipulating terms with free λ-variables.

Universal quantification in goals is an indirect means for handling structural induction through λ-abstractions. Then η-equivalence in the equality theory is required for making the interpretation of λ-abstraction by universal quantification correct and reversible (see the conservation condition in Section 3.2.2). Finally, for structural induction definitions to remain complete with respect to types when universal quantification introduces new constants, implication is used for extending the induction rules and handling the new constants.

Our reconstruction does not deal with disjunction in goals, $\vee_{\mathscr{G}}$, and with existential quantification in goals, $\exists_{\mathscr{G}}$. Though they do not really belong to Horn formulas, their right-introduction rules are representable by second-order Horn clauses.

```
type ';' o -> o -> o.
type sigma (T-> o) -> o.

G1 ; G2 :- G1.      G1 ; G2 :- G2.      %  P ⊢ Gi / P ⊢ G1∨G2      ∨_G (i.e., ';')

sigma G :- (G T).                       %  P ⊢ (G t) / P ⊢ ∃x (G)   ∃_G (i.e., sigma)
```

So, they are not difficult at all, and most Prolog systems feature $\vee_{\mathscr{G}}$. They are mostly useful for building complex formulas without using intermediate predicates.

Among the fragments of λProlog that we have presented in the introduction, CLP($\lambda_{\rightarrow}$) and αβProlog do not conform to our criterion because they feature components that are high in Fig. 1 (λ-terms and αβ-equivalence) without featuring the components that are below them (i.e., that make them possible).

We analyze now the status of Typed Prolog and Harrop Prolog. As opposed to other fragments, the features they lack are high in Fig. 1. Typed Prolog seems a good approach to programming in first-order λProlog, because this dialect invites the programmer to describe the data-structures more precisely than what is usually done in Prolog. This is an important point of the proposed programming method.

The language Harrop Prolog offers connectives $\forall_{\mathscr{G}}$ and $\Rightarrow_{\mathscr{G}}$, but does not feature λ-abstraction. It will miss λ-terms for meta-programming. Prolog does without higher-order terms by using either the *ground* representation technique, representing object-level variables by constants, or the *non-ground* representation technique, representing object-level variables by meta-level variables. The former is usually sound, but often cumbersome, whereas the latter is simpler, but unsound. It may work when the quantification structure is simple, as in Horn formulas, but it becomes laborious when the quantification structure is as complex as with Harrop formulas. However, Harrop Prolog is still valuable for other programming idioms like hypothetical queries (see predicate *presumed_grandfather* in Section 2.2.4), abstract data-types, or memoing (see a glimpse of it at the end of Section 5.3).

There are other global presentations of λProlog. Miller and Nadathur give a proof-theoretic overview of λProlog [48], but it contains little practical motivations for all the features of λProlog. The presentation of "extended natural semantics" by Hannan [20] can be seen as a reconstruction of λProlog whose initial motivation is to build specifications *via* deduction rules. He motivates every feature of λProlog by the need for representing higher-order abstract syntax, traversing it, and representing an environment for the deduction. Our reconstruction is original in that its puts the activity of programming in a central position. It rephrases for the programmer elements that were scattered in λProlog's literature and sometimes beyond. It also offers new points of view on the relation with the theory of inductive types and on the role of η-equivalence in the conservation condition.

5.2. L_{λ}

5.2.1. The principles of L_{λ}

Miller proposes a fragment of λProlog, L_{λ}, that restricts more strongly the term domain than simple types do. In this fragment, the unification problem is decidable and unitary [34]. The key idea is to substitute for axiom β, which is difficult to reason with, a weaker, but easier, axiom.

Axiom β_0**:** $(\lambda x(E)y) =_{\beta_0} E[x \leftarrow y]$, if $y \in \mathscr{V}$ and $y \notin \mathscr{FV}(\lambda x(E)) \cup \mathscr{BV}(E)$.

This axiom formalizes a weak form of β-equivalence where the actual parameter is restricted to be a variable which is neither free in the λ-abstraction, nor bound in its body.

The β_0-reduction of $(\lambda x(E)\ y)$ amounts to renaming variable x into y in E. This can be shown as follows. If a variable y satisfies the precondition of axiom β_0, then it satisfies the precondition of axiom α (see Section 2.1.3). The α-conversion of $\lambda x(E)$

into $\lambda y(E[x \leftarrow y])$ keeps the precondition of axiom β_0 true because $y \notin \mathscr{FV}(\lambda x(E)) \cup \mathscr{BV}(E)$ implies $y \notin \mathscr{FV}(\lambda y(E[x \leftarrow y])) \cup \mathscr{BV}(E[x \leftarrow y])$, and it makes the computation of $E[x \leftarrow y]$ trivial because x would be already equal to y.

Any two β_0-equivalent terms are also β-equivalent, but the opposite is false. Considering arbitrary subsets of the λ-terms, there may be pairs of β-equivalent terms that are also β_0-equivalent, and others that are not. By definition, the term domain of L_λ is the greatest subset of the λ-terms for which β_0-equivalence is equal to β-equivalence: any two β-equivalent terms are also β_0-equivalent.

This domain can be characterized syntactically by a restriction of the rule for building applications: an existential variable can only be applied to distinct λ-variables and distinct universal variables that are quantified in its scope. For instance, $\lambda x \lambda y \lambda z(U\ x\ z)$ is a L_λ-term because the existential variable U is applied to distinct λ-variables. On the opposite, $\lambda x \lambda y \lambda z(U\ V)$, $\lambda x \lambda y \lambda z(U\ x\ x)$ and $\lambda x \lambda y \lambda z(U\ [x])$ are not L_λ-terms because the existential variable U is applied to another existential variable, or it is applied to two identical λ-variables, or it is applied to a compound term. More subtly, $\lambda x(U\ x)$ is a L_λ-term in $\exists U \forall x[\ldots \lambda x(U\ x)\ldots]$, but not in $\forall x \exists U[\ldots \lambda x(U\ x)\ldots]$, because x is not quantified in the scope of U.

$L_\lambda$is enough for many applications, but cannot be used for coding

$$\mathscr{T}_g[\![B_1, B_2]\!] = \lambda\kappa.(\mathscr{T}_g[\![B_1]\!]\ (\mathscr{T}_g[\![B_2]\!]\ \kappa))$$

as

$$t\_g\ (B1\ and\ B2)\ \lambda k\underline{(D1\ (D2\ k))} \Leftarrow\ t\_g\ B1\ D1\ \wedge t\_g\ B2\ D2$$

(as is done in Section 3.1.1) because the underlined term does not belong to L_λ. Predicates *beta_conv* (Section 2.1.3) and *sigma* (Section 5.1) do not belong to L_λ either: terms $(E\ F)$ and $(G\ T)$ are illegal. More generally, higher-order programming (where functions are applied to arbitrary terms, and not only to distinct universal variables and λ-variables) is forbidden in L_λ. However, all the examples of structural induction given in this article belong to L_λ.

Though many useful programs do not belong to L_λ, Miller shows that every λProlog program can be translated into a L_λ program [38]. Even when not used as the kernel of λProlog as Miller proposes, the L_λ fragment can be used as a heuristic criterion to avoid using the general but costly unification procedure [6,5].

5.2.2. The structure of L_λ

One may wonder if the L_λ fragment of λProlog is restricted enough for modifying the overall structure of the language.

In fact, $L_\lambda$has exactly the same structure as λProlog. Unification in L_λ still does not allow us to analyze every data-structure. The restriction is such that even application $(T_1\ T_2)$ cannot be formed in L_λ. So, one still needs universal quantification, η-equivalence, and implication. In fact, the translation of λProlog into L_λ [38] is an illustration of this.

The idea of this translation is to encode the theory of β-equality in L_λ formulas, and to modify the program so that it uses the encoded theory instead of the built-in theory of λProlog. However, the encoding itself is an example of the relation between formulas and terms that we have exposed. It uses abundantly connectives $\forall_g$ and $\Rightarrow_g$ for defining equality modulo $\alpha\beta$-equivalence by structural induction. For instance, if there is a constant h of type $(i \rightarrow j) \rightarrow k$ in the program, the encoding yields the clause

copy_k (h X) (h U) :– pi y\(pi v\ (copy_i y v => copy_ j (X y) (U v))).

$\%\forall$ y v[copy$_i$ y v $\Rightarrow$ copy$_j$ (X y) (U v)] $\Rightarrow$ copy$_k$ (h X) (h U)

for specifying equality of two terms whose main constructor is h. Since the transformation considers all types at the same time, relations *copy_*τ are defined by mutual recursion for all base types τ. This makes every negative occurrence of a type, here i, the cause of the non-inductiveness of this type. The arguments, X and U, of the two h's are of a function type, $(i \rightarrow j)$, and universal quantifications *pi* $y\backslash$ and *pi* $v\backslash$ are used to propagate the *copy* relation trough them, $(X\ y)$ and $(U\ v)$. Since y and v are of the non-inductive type i, the definition of *copy_i* is extended in their scope. In this case, the extension to the definition says that y and v must be considered $\alpha\beta$-equivalent.

Note that terms $(X\ y)$ and $(U\ v)$ belong to L_λ, and that X and U, which are of function type, are not directly compared for testing equality but only their results when applied to y and v. This is correct only if axiom η is in the encoded theory. Otherwise, the fact that $(X\ y)$ and $(U\ v)$ are equal for every y and v (assumed equal) would not imply that X and U are also equal.

5.3. Idioms for programming

Our reconstruction shows that there is no "interesting" system between Typed Prolog and λProlog (or L_λ) for programming by structural induction on λ-terms. This would be bad news if it condemned the user to tackle at once all the complexity of λProlog. However, a programmer can still enter progressively in λProlog. The first step is to use it as Typed Prolog. The next step is to program by structural induction on inductive data-structures; this only involves $\forall_\mathcal{G}$. The last step is to program with non-inductive data-structures; this brings in $\Rightarrow_\mathcal{G}$. The beginner can introduce in parallel other programming idioms that do not require a global understanding of λProlog. Some of them are recalled at the end of this section.

The complexity of λProlog stems from a greater variety and a greater precision in logical phrases. In Prolog, this complexity does not appear in the language, but it appears in the programs that need to encode in rudimentary terms the desired precision. It happens that many of the encodings that are practically used in Prolog (e.g., non-ground representation of object-level structures) are not correct if one considers the declarative semantics of Prolog (see Section 3.1.1).

The programming discipline presented in this article considers λ-abstraction and universal quantification in goals symmetrically. This important point for programming is illustrated in Fig. 2. λ-Abstraction is used as a *reified* form of universal quantification, the latter serving as a *reflection* of the former.

For analyzing/consuming/reflecting a λ-abstraction, one must apply it to a universal variable whose scope is included in the scope of the variable that is bound to the λ-abstraction. For synthesizing/producing/reifying a λ-abstraction, one must build a term in which a unique universal variable represents every occurrence of the variable

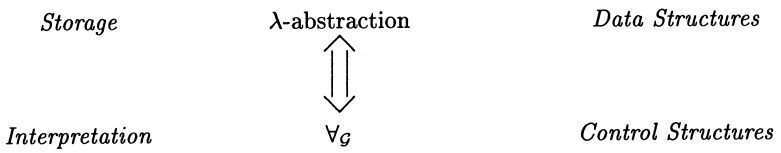


Fig. 2. λ-abstraction vs. universal quantification.

bound by the λ-abstraction. It represents the body of the λ-abstraction. The λ-abstraction itself is computed as the function that yields that term when it is applied to the same universal variable (see for instance the proof of (*nnf* (*exists x*(*not* (*exists y*(*p x y*)))) *X*), Section 3.4.2).

This article does not cover every way of programming in λProlog. λProlog programming with first-order terms often uses techniques similar to Prolog's, with typing added. However, there is still a possibility to use implication in goals. The following version of the *append* predicate is an illustration of this possibility.

```
type append (list T) -> (list T) -> (list T) -> o.
type append0 (list T) -> (list T) -> o.
type final (list T) -> o.

append0[ ] L2 :- final L2.
append0 [A | L1] [A | L3] :- append0 L1 L3.
append L1 L2 L3 :- (final L2 => append0 L1 L3) .
```

Here, implication is used for making term *L2 global*. This shows how a context can be transmitted through a program clause rather than through a term. In this case, implication is not motivated by the structure of a term, but rather by an invariant of the computation. A variant of this is a memoing idiom where implication is used to "store" intermediate results that are "fetched" during a computation.

Functional data-structures are another important idiom. They play in λProlog a role similar to incomplete data-structures in Prolog (e.g., difference lists), but they allow for a more declarative programming style [7].

Our reconstruction insists on a symmetry between λ-abstraction and universal quantification in goals. This may not be the case in other idioms. For instance, when the positions of λ-abstractions are predetermined, one can go through them simply using higher-order unification like in the following derivation program.

```
type derivative (R-> R) -> (R-> R) -> o.

derivative x\ x x\ 1.
derivative x\ Const x\ 0.
derivative x\ ((A x) + (B x)) x\ ((DA x) + (DB x)) :-
   derivative A DA, derivative B DB.
```

6. Conclusion

We have presented a demand-driven reconstruction of the various features of λProlog. The initial demand is to be able to program by structural induction on λ-terms. A first observation is that to ensure that unification is well-defined and tractable, the domain of λ-terms must be restricted in some way; the theory of simple types is such a way. A second observation is that universal quantification in goals and η-equivalence in the equality theory are required for expressing relations between λ-abstractions and parts of their bodies. The last observation is that implication in goals is needed to allow for structural induction definitions of predicates where

the signature is dynamically extended by the interpretation of universal quantification in goals.

A crucial point is that the λ-terms manipulated by λProlog must be combinators. This observation is independent from our reconstruction, but has seldom been commented on by the designers of λProlog, though we think it is of the greatest importance for the programmer.

The combined requirements of logic programming (reversibility) and of structural induction (relating λ-abstractions and their bodies) have shown a symmetric correspondence between λ-abstractions and universal quantifications in goals. It is η-equivalence that makes it fully symmetric.

This correspondence between terms and formulas inspires a programming method by which a programmer first designs a data-structure and the types of its constructors. Then, the programmer uses these types as a guide for defining relations. If a constructor has a first-order type, a Prolog-like structural induction will do. Constructors of higher-order type will need universal quantifications in goals, and among them, constructors whose type contains negative occurrence of the induction type will also need implication in goals. This method could be embodied in a tool that, given the declaration of the constructors of a data-structure, would produce the iterator of this data-structure. Then the programmer would have to fill in the blanks to produce an actual program. One can even imagine that the blanks be filled in semi-automatically by using techniques such as inductive logic programming[6] [51] and higher-order unification [19]. Further refinements are needed for making the scheme more flexible, for handling mutually recursive definitions, and for taking into account polymorphism, but we think it covers a programming idiom that is frequently used.

Acknowledgements

We like to thank the anonymous referees for their helpful comments, and Dale Miller and Gopalan Nadathur for inventing such an exciting programming language.

References

[1] H. Andréka, I. Németi, The generalised completeness of Horn predicate-logic as a programming language, DAI Research Report 21, University of Edinburgh, 1976; Acta Cybernetica 4 (1978) 3–10.
[2] H. Barendregt, Introduction to generalized type systems, J. Functional Programming 1 (2) (1991) 125–154.
[3] C. Belleannée, Vers un démonstrateur de théorèmes adaptatif, Thèse, Université de Rennes 1, 1991.
[4] C. Böhm, A. Berarducci, Automatic synthesis of typed λ-programs on term algebras, Theoretical Computer Science 39 (1985) 135–154.
[5] P. Brisset, O. Ridoux, The architecture of an implementation of λProlog: Prolog/Mali, in: Workshop on λProlog, Philadelphia, 1992, Revised version in ILPS Workshop on Implementation Techniques for Logic Programming Languages, 1994.

[6] In this case, *inductive* is meant as opposed to *deductive*.

[6] P. Brisset, O. Ridoux, The compilation of λProlog and its execution with MALI, Publication Interne 687, IRISA, 1992.
[7] P. Brisset, O. Ridoux, Naïve reverse can be linear, in: K. Furukawa (Ed.), 8th Int. Conf. Logic Programming, MIT Press, 1991, pp. 857–870.
[8] A. Church, A formulation of the simple theory of types, J. Symbolic Logic 5 (1) (1940) 56–68.
[9] J. Cohen, Constraint logic programming languages, CACM 33 (7) (1990) 52–68.
[10] B. Coüasnon, P. Brisset, I. Stéphan, Using logic programming languages for optical music recognition, in: A. Marien (Ed.), Third Conf., The Practical Application of Prolog, Alinmead Software Ltd, Paris, France, 1995, pp. 115–134.
[11] M. Dalrymple, S.M. Shieber, F.C.N. Pereira, Ellipsis and higher-order unification, Linguistics and Philosophy 14 (1991) 399–452.
[12] N.G. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, Indagationes Mathematicae 34 (1972) 381–392.
[13] C.M. Elliott, Higher-order unification with dependent function types, in: N. Derschowitz (Ed.), Third Int. Conf. Rewriting Techniques and Applications, LNCS 355, Springer, Berlin, 1989, pp. 121–136.
[14] C.M. Elliott, F. Pfenning, A semi-functional implementation of a higher-order logic programming language, in: P. Lee (Ed.), Topics in Advanced Language Implementation, MIT Press, 1991, pp. 289–325.
[15] A. Felty, Implementing tactics and tacticals in a higher-order logic programming language, J. Automated Reasoning 11 (1) (1993) 43–81.
[16] A. Felty, Specifying and implementing theorem provers in a higher-order logic programming language. Ph.D. Dissertation, Dept. of Computer and Information Science, University of Pennsylvania, 1989.
[17] J.H. Gallier, Logic for Computer Science: Foundations of Automatic Theorem Proving, Harper and Row, 1986.
[18] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1987) 1–102.
[19] M. Hagiya, Synthesis of rewrite programs by higher-order and semantic unification, New Generation Computing 8 (4) (1991) 403–420.
[20] J. Hannan, Extended natural semantics, J. Functional Programming 3 (2) (1993) 123–152.
[21] J. Hannan, D. Miller, From operational semantics to abstract machines, Mathematical Structures in Computer Science 4 (2) (1992) 415–459.
[22] R. Harrop, Concerning formulas of the types A→B∨C, A→(Ex)B(x) in intuitionistic formal systems, J. Symbolic Logic 25 (1) (1960) 23–27.
[23] J.S. Hodas, D.A. Miller, Logic programming in a fragment of intuitionistic linear logic, Information and Computation 110 (2) (1994) 327–365.
[24] A. Horn, On sentences which are true of direct unions of algebras, J. Symbolic Logic 16 (1) (1951).
[25] W.A. Howard, The formulae-as-types notion of construction, in: J.P. Seldin, J.R. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, London, 1980, pp. 479–490.
[26] G. Huet, A unification algorithm for typed λ-calculus, Theoretical Computer Science 1 (1975) 27–57.
[27] T.K. Lakshman, U.S. Reddy, Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system, in: V. Saraswat, K. Ueda (Eds.), Eighth Int. Logic Programming Symp., MIT Press, 1991, pp. 202–217.
[28] S. Le Huitouze, P. Louvet, O. Ridoux, Logic grammars and λProlog, in: D.S. Warren (Ed.), 10th Int. Conf. Logic Programming, MIT Press, 1993, pp. 64–79.
[29] C.C. Liang, Object-level Substitution, Unification and Generalization in Meta-Logic, Ph.D. Thesis, University of Pennsylvania, Department of Computer and Information Science, 1995.
[30] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1987.
[31] D. Miller, A multiple-conclusion meta-logic, in: Symp. Logic in Computer Science, 1994, pp. 272–281.
[32] D.A. Miller, Abstract syntax and logic programming, in: A. Voronkov (Ed.), Second Russian Conf. Logic Programming, LNCS 592, Springer, Berlin, 1991.
[33] D.A. Miller, Lexical scoping as universal quantification, in: G. Levi, M. Martelli (Eds.), 6th Int. Conf. Logic Programming, MIT Press, 1989, pp. 268–283.
[34] D.A. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, in: P. Schroeder-Heister (Ed.), Int. Workshop on Extensions of Logic Programming, LNAI 475, Springer, New York, 1989; Also in J. Logic and Computation 1 (4) (1991) 497–536.

[35] D.A. Miller, A logical analysis of modules in logic programming, J. Logic Programming 6 (1/2) (1989) 79–108.
[36] D.A. Miller, A proposal for modules in λProlog, in: R. Dyckhoff (Ed.), Int. Workshop Extensions of Logic Programming, LNAI 798, Springer, 1993, pp. 206–221.
[37] D.A. Miller, A theory of modules for logic programming, in: Symp. Logic Programming, Salt Lake City, UT, USA, 1986, pp. 106–115.
[38] D.A. Miller, Unification of simply typed lambda-terms as logic programming, in: K. Furukawa (Ed.), Eighth Int. Conf. Logic Programming, MIT Press, 1991, pp. 255–269.
[39] D.A. Miller, Unification under a mixed prefix, J. Symbolic Computation 14 (1992) 321–358.
[40] D.A. Miller, G. Nadathur, Higher-order logic programming, in: E. Shapiro (Ed.), Third Int. Conf. Logic Programming, LNCS 225, Springer, 1986, pp. 448–462.
[41] D.A. Miller, G. Nadathur, A logic programming approach to manipulating formulas and programs, in: S. Haridi (Ed.), IEEE Symp. Logic Programming, San Francisco, CA, USA, 1987, pp. 379–388.
[42] D.A. Miller, G. Nadathur, F. Pfenning, A. Scedrov, Uniform proofs as a foundation for logic programming, Annals of Pure and Applied Logic 51 (1991) 125–157, Revised version of Ref. [43].
[43] D.A. Miller, G. Nadathur, A. Scedrov, Hereditary Harrop formulas and uniform proof systems, in: D. Gries (Ed.), Second Symp. Logic in Computer Science, Ithaca, NY, 1987, pp. 98–105.
[44] R. Milner, A theory of type polymorphism in programming, J. Computer and System Sciences 17 (1978) 348–375.
[45] R. Montague, The proper treatment of quantification in ordinary English, in: R.M. Thomason (Ed.), Formal Philosophy, Yale University Press, New Haven, CO, 1974.
[46] G. Nadathur, A higher-order logic as the basis for logic programming, Ph.D. Thesis, University of Pennsylvania, 1987.
[47] G. Nadathur, D.A. Miller, Higher-order Horn clauses, JACM 37 (4) (1990) 777–814.
[48] G. Nadathur, D.A. Miller, An overview of λProlog, in: K. Bowen, R. Kowalski (Eds.), Symp. Logic Programming, Seattle, WA, 1988, pp. 810–827.
[49] L. Naish, Negation and Control in Prolog, LNCS 238, Springer, Berlin, 1986.
[50] T. Nicholson, N. Foo, A denotational semantics for Prolog, ACM Trans, Programming Languages and Systems 11 (4) (1989) 650–665.
[51] S.-H. Nienhuys-Cheng, R. de Wolf, Foundations of Inductive Logic programming, LNAI 1228, Springer, Berlin, 1997.
[52] R. Pareschi, D.A. Miller, Extending definite clause grammars with scoping constructs, in: D.H.D. Warren, P. Szeredi (Eds.), Seventh Int. Conf. Logic Programming, MIT Press, 1990, pp. 373–389.
[53] C.S. Peirce, in: C. Hartshorne, P. Weiss (Eds.), Collected Papers of Charles Saunders Peirce, 2nd ed., Harvard University Press, 1960.
[54] F. Pfenning, Unification and anti-unification in the calculus of constructions, in: Symp. Logic in Computer Science, 1991, pp. 74–85.
[55] B. Pierce, S. Dietzen, S. Michaylov, Programming in higher-order typed lambda-calculi, Research Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, 1989.
[56] O. Ridoux, Engineering transformations of attributed grammars in λProlog, in: M. Maher (Ed.), Joint Int. Conf. and Symp. Logic Programming, MIT Press, 1996, pp. 244–258.
[57] O. Ridoux, Imagining CLP($\lambda, \equiv_{\alpha\beta}$), in: A. Podelski (Ed.), Constraint Programming: Basics and Trends, Selected papers of the 22nd Spring School in Theoretical Computer Science, LNCS 910, Springer, Châtillon/Seine, France, 1995, pp. 209–230.
[58] L. Sterling, E. Shapiro, The Art of Prolog, 1st ed., MIT Press, 1986.
[59] S.-Å. Tärnlund, Horn clause computability, BIT 17 (1977) 215–226.